Proof Planning with Logic Presentations

Santiago Negrete-Yankelevich

Ph.D.
University of Edinburgh
1996

Abstract

Logic has proven to be of prime importance in many areas of Science and, in particular, in AI and Computer Science. Many logics have been developed to contend with the various kinds of reasoning required by a vast number of research areas.

Automatic theorem proving techniques for these logics are often developed in an ad-hoc way for particular theories in which specific problems are representable. *Framework logics* have been proposed as meta-mathematical theories in which other theories may be represented and reasoned with uniformly. Hence, automating proof search in a framework logic gives the possibility of abstracting the proof process and make it applicable to a larger number of logics.

In this thesis, we introduce an approach to proof search in the Edinburgh Logical Framework that is not hard-wired to a particular object logic. The design is based on guiding proof search through constrained rewriting. The rewriting technique is called *Rippling*; it has been previously applied to the domain of inductive proofs within the framework of Proof Plans. This approach consists of building proof plans for theorems from abstract specifications of proof techniques called Methods. Methods correspond to tactics programmed in a Proof Editor where the theorem can be represented and where a proof plan can be realized to construct the actual proof for the theorem.

Our work is focussed on logic presentations and we have extended the Rippling technique in various ways to handle this more general case. A feature of our approach is that, since the rewrite rules used in Rippling are extracted from the logic presentation in the framework logic, the proof mechanisms are independent of the logic at hand.

We have conducted experiments in various classical and non-classical logics and conjecture that our methodology is applicable to frameworks and logics other than those we have explored here.

Acknowledgements

I would like to thank foremost my supervisors: Dr. Alan Smaill, Dr. Ian Green and Dr. Ian Gent for their support, advise and encouragement throughout my work on this thesis.

I want to thank Prof. Alan Bundy and all the members of the DReaM group at Edinburgh who contributed with ideas, discussions and in many other ways to this thesis as well as providing a very special research environment.

Special thanks are due to David Basin, Francisco Cantú, Ina Kraan and Pablo Noriega who read earlier versions of this thesis and gave me important comments and suggestions.

John Ayscough, Silvana Parascandolo and "the family" provided me with a home where I've lived happily during the final stages of the write up. I will always be in debt with them.

Special thanks go to my family who have always supported what I do and are a constant source of encouragement and inspiration.

Peppe, Marina and Mariano from cafe *Sorriso* provided important Mediterranean environment, coffee and good humour during the lowest points of winter. Many parts of this thesis were generated thanks to these ingredients.

It is difficult to thank individually all my friends in Edinburgh who made of this city a very nice place to live in and a very difficult place to leave. They all know who they are.

I dedicate this thesis to Grace for the good and hard times we've shared.

I gratefully acknowledge the financial support for this work given by D.G.A.P.A. from Universidad Nacional Autónoma de México.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

S. Negrete Y.
Edinburgh
April 1, 1996

Contents

List of Figures

Chapter 1

Introduction

In this thesis we propose a new approach to generic theorem proving. It consists of a parameterisation of the prover by logic presentations. The prover extracts proof tools from a given logic and uses them to automate search in that logic.

This Chapter is the introduction to the thesis. The first sections constitute the motivations for the work. Section 1.4 contains a survey of the project and the final section presents an outline of this document.

1.1 Mathematics for Computing

The second half of the 20th century has seen the rise of the computer from a mathematical object in the 30's through big artifacts with little power in the 60's, up to powerful machines, encased in small boxes, ubiquitous in our daily life and as common to us as cars, in the 90's.

Even before the creation of the first computer, the concept of *computation* has been a subject of research. Over the years, computer science research has produced several theories to describe this phenomenon and, from these, a long succession of computer languages emerged.

The Turing machine was a near description of the actual workings of a possible machine, but from then on, formalisms have become more and more abstract in order to shorten the gap between the machine's "levers and pulleys" and human speech and thought. We strive to relieve ourselves from the burden of writing long streams of hexadecimal

numbers and to be able to communicate with computers in a way more akin to human experience. In order to bridge this gap between machines and common citizens, we use mathematical theories, powerful enough to embrace Computation but also suitable for human problem-solving reasoning.

As these theories become more elaborate, we delegate the translation of our algorithms to the computer itself by building compilers. The process of translating an algorithm from specification to machine code, originally done manually by programmers in a long and error-prone way, becomes in itself an algorithm built into the computer. This way, computer software becomes more complex and relies on previously programmed systems.

In the 70's, software reliability was being reinforced by introducing the programmer to programming disciplines and techniques that reduced the probability of errors. These techniques like structured programming and software management, seemed to cope with the tasks at hand. In recent years however, not only computer systems (software as well as hardware) have become larger and more complex, but also our dependency on them has increased. We no longer can afford computer malfunction when human life is involved. The need to make these complex systems reliable is one of the main problems in computer science in this final decade of the century.

Once more, mathematical theories have been proposed to address the problem. Formal specification languages are being developed so that program specifications can be proven correct before they are actually implemented. Unfortunately, the complexity of the specifications has made it prohibitive for a human to prove manually and in a reasonable time/cost the specifications of most software and hardware systems of industrial strength. What we need is a machine to prove them.

1.2 Computing for Mathematics

Science has always caused the creation of new tools, either because new ideas bring new products into the market or because scientific research requires new tools to work. Mathematics has promoted the creation of tools like rulers, calculators, slide rules, etc. and, although these tools are now largely considered useful in areas other than

mathematics, it is only due to the specialisation in science brought in by our century that distinctions of this kind have been made.

In the Renaissance, scientists spent long nights performing calculations to check their predictions. These scientists needed calculating machines because the calculations they did took too long and were not reliable. Their life time was often not long enough to accomplish the goals they had set to themselves with the resources they had.

Today, our tools are more sophisticated but so have become our scientific enterprises. Supercomputers calculate complex systems of equations in seconds and other hard but tedious tasks and thus free our minds to address more intellectually challenging problems. Even so, there are tasks today which, even though they don't involve numeric computation, do require mechanical symbolic processing.

Using computers as assistants in the more abstract processes of mathematics is an appealing idea. These may include symbolic manipulations, like derivation, integration, equation-solving and logical inference. This thesis is concerned with the automation of the last of the above. We would like to develop computer systems that help mathematicians, computer scientists, and other researchers, by doing accurate and complex inference operations in different theories. This way the user would not need to worry about writing or examining routine proofs, verifying inference steps or the applicability of lemmas, etc. and could devote all her efforts to the inventive aspects of her discipline.

To write a computer program to perform any task, we need to have a clear idea of what we want to achieve and so our enterprise requires an understanding of how Mathematics and mathematicians work.

Framework Theories [Harper *et al* 92, Constable *et al* 86, Coquand & Huet 88] are formal theories designed to represent other formal systems' syntax and inference. They account for many aspects of inference —common to most theories— in a uniform way. These aspects include substitution, variable binding, context handling. The objects the framework theories deal with are meta-theoretical concepts of other theories called the *object theories*. Therefore, proof automation in framework theories implies proof automation in any object theory representable in the framework.

There are a number of computer system where formal theories can be represented and theorems proven by interactively telling the system the order of the inference rules to apply. These systems, called *proof editors* [Constable *et al* 86][Luo & Pollack 92][Richards *et al* 94] perform all the necessary operations for each inference rule application. Some of these systems are based on framework theories and some provide a language to write computer programs —called *tactics*— that combine inference rule applications enabling the user to apply bigger proof steps. In Section 2.2 we analyse these systems.

Proof editors can be used as the basis to develop proof search automation techniques to solve complex mathematical problems.

1.3 Proof Search in Framework Theories

The problem of proof automation in general is undecidable. Therefore, all we can aim at is automating specific aspects of it: theorem families, specific theories, etc. We also want to learn about the nature of theorem proving, what is it that we do when we prove theorems, what are the creative sides of it and what are the "mechanical" ones.

In this spirit, we see theorem proving as a long term enterprise. New theories will come with new topics and our systems will have to automate search in these theories. We want therefore to build a system with the versatility and capability of withstanding this evolution. Our system must both be able to provide a setting for experimentation with proof automation techniques in new theories and adapt older techniques to the new cases. It must be able to produce proofs where proof steps are intelligible to the user. This property is essential for a system where experimental techniques are to be tested because in such a system successful proofs are just as useful as unsuccessful ones. The user needs to know what went wrong in a failed proof attempt in order to either revise the underlying theory or strengthen the proof automation.

Inference at the framework level has finer granularity with respect to the object theory than direct inference. To each inference step of the object theory correspond several inference steps at the framework level. This feature allows a more detailed reasoning of the proof process at the object level but makes approaches to proof automation based

on uniform search at the level of the framework logic very hard to control. A system that controls search effectively over a big range of logics will have to obtain guidance from each particular logic.

Logic Programming-based systems [Pym 90],[Pfenning 91], [Felty & Miller 91] obtain guidance from the object level through unification. This approach is good as a basis for tactic theorem proving but does not constrain search sufficiently on its own.

1.4 Proof Planning with Framework Theories

In this thesis we take a new approach to generic proof search. We develop a system that obtains guidance from the object level logic by extracting rewrite rules from each logic presentation. The rewriting process is guided by a general strategy based on a special type of unification. The system obtained is parameterised by the logic presentations of the framework logic and is therefore able to do a more specialised search guidance in each logic than a system based on uniform search.

1.4.1 Proof Plans

We use *Proof Plans* [Bundy *et al* 91] as a paradigm to develop proof-automation techniques. Proof Plans consists of using a planner to build a proof plan from specifications of tactics called *Methods*. Methods allow us to separate the proof building procedure from the reasoning required to select a proof technique. Writing methods amounts to specifying when a proof technique should be used as opposed to verifying if the technique is applicable, which is what a proof editor does when it tries to apply a tactic to the current state.

Building proof plans from methods has the advantage of not having to do all the operations needed to apply the tactics. The architecture of the methods enables a declarative specification of the heuristics for the tactics. This way, it is easy to edit and experiment with the heuristic information without altering the tactics themselves.

The methods we present in this thesis are designed for proof planner called MiniClam which is related to Clam [vanHarmelen *et al* 93]. The proof plans built by MiniClam

are applicable in a version of the Edinburgh Logical Framework [Harper *et al* 92] programmed in the Mollusc proof editor [Richards *et al* 94].

1.4.2 Difference Reduction

Logics are usually represented in proof editors as a set of axioms and inference rules. Inference rules are the tools to convert one proof state into another and axioms just tell us when to stop —when building a proof backwards (from theorem to axioms)— or where to start —when building it forwards (from axioms to theorem).

In our system, proofs are built backwards. Axioms are sequents —in the sense of the Sequent Calculus [Gentzen 69]— where the formula on the right-hand side (i.e. the goal) occurs in the left-hand side (i.e. the context). To obtain these from the conjecture, one must decompose it by applying inference rules. Each inference rule application transforms one proof state into another. Guiding search in a proof consist of deciding what inference rule is the most appropriate to transform every state into another one closer to an axiom.

The proof plans built by our system are based on rewriting operations as well as single inference rule applications. The operations are tactics that apply the appropriate inference rules to produce the desired effect in the proof.

In order to select each rewriting operation the system uses the principle of *Difference Reduction* [Basin & Walsh 96] which consists of two steps:

1. Compare two expressions using a special unifier called *difference unifier* [Basin & Walsh 93] to obtain annotations over the expressions. These annotations indicate the parts of the expressions that would have to be removed (the difference) in order to unify them.

2. Use rewrite rules to successively rewrite the differing expressions in such a way that the difference between the two is reduced at each step.

To monitor the movement of the difference, indicated in the original expression by the annotations, we annotate the rewrites as well. The annotations in the rewrites

describe how the difference moves through the rule and help us to select the rules that will move the difference in the right direction. The process continues until no more difference reduction can be done. Annotations also permit the definition of a measure of annotated terms. Wave rules are all by construction measure-decreasing rewrite rules and hence rippling is guaranteed to terminate.

In our case, since we are interested in transforming a sequent into an axiom, we compare the goal and the hypotheses of sequents. The rewriting operations selected by the planner are those that reduce the difference between the goal and the hypothesis closest to it in terms of similarity.

The annotated rewrite rules are called *wave rules* and the process of successive rewriting using wave rules is called *rippling*. The version of rippling we use in this thesis is an extension of the original rippling developed for inductive proofs [Bundy *et al* 93].

Rippling selects wave rules by matching their left-hand sides with the target expression in the usual way for rewrite rules; however, matching with annotations guarantees that the appropriate structures of the target expression will be shifted. Successive rewriting with wave rules may lead to the complete elimination of the difference between induction hypotheses and conclusion and hence to an axiom. This form of rewriting is more constrained than the unannotated kind and increases the chances of attaining a desired state.

The intuition behind our project is that we can view the search for an axiom analogous to the search for fertilisation in inductive theorem proving (Section 3.3) (i.e. when the induction hypothesis can be used to prove the induction conclusion because they can be unified). Inference rules of encoded logics in the framework can also be seen as rewrite rules that can be applied in two directions: left to right when refining the goal with the rule and right to left when forward chaining the rule with a hypothesis. These comparisons with the inductive case suggest similar techniques in the context of framework theories.

An important concept in our work is that of polarity. We use polarity annotations on framework theory expressions to make sure that rewriting operations in the plan correspond to sound inference rule applications in the proof editor. We also annotate

object-logic expressions with polarity annotations derived from the corresponding signature heuristic to restrict difference unification to focus only on expressions that are likely to produce axioms.

We have extended many notions used in rippling to deal with framework logics: annotated term, wave rule, term rewriting and polarity. In Chapter 4 we discuss these topics in detail and explain how they fit together.

Difference reduction provides a pattern to develop different search strategies. By alternating difference unification and control techniques to reduce differences, we can build general methods to plan proofs in frameworks theories. The preferred control technique is rippling because is the most constrained. After rippling, unannotated rewriting is attempted. Finally, if the two previous options are not successful, inference rules are applied directly.

This hierarchy means that in the best cases, when only rippling steps are used, the search required to prove a theorem will be very small. In the places where rippling does not apply, the system may resort to more expensive steps to continue with the proof and try to resume rippling. This way, our system applies a well constrained methodology, like rippling, to produce proofs without much search but, when the methodology is not appropriate to a particular case, it *gracefully* degenerates into unconstrained search. From this point of view, our system is a hybrid approach to generic proof search guidance between specific systems with little search and scope, and uniform proof search methods which are very general but produce big search spaces.

The methods designed using these techniques have been tested by hand with theorems from propositional, predicate and modal logics, both intuitionistic and classical. We also report on an implementation of an earlier version of the methods on which the first experiments where carried out.

1.5 Thesis Outline

The thesis has the following chapters. The first two constitute the background knowledge of the thesis. Chapter 2 describes the concepts of framework theory and proof editor giving a list of the best known instances of them. Chapter 3 describes the basic

vocabulary and concepts related to proof plans and rippling. Chapter 4 presents an overview of the techniques of the thesis. There, all the concepts used in the rest of the thesis are outlined and their relationships explained so that reading the next chapters becomes easier for the reader. An example is also given in Section 4.3.

After the overview chapter, the main techniques are explained in detail. Chapter 5 gives definitions and properties of polarity. Chapter 6 has all the definitions relevant to annotated terms and the algorithm for difference unification. Chapter 7 describes how to obtain rewrite rules from a signature and wave rules from rewrite rules. It also explains annotated term rewriting. This concludes the technical description.

Chapter 8 is a description of the design of the system in terms of methods and submethods. Chapter 9 presents the example logics and theorems used to test the system described in Chapter 8 and, finally, in Chapter 10, conclusions are drawn from the thesis and prospects for future work are examined.

Chapter 2

Framework Theories and Proof Editors

This chapter is the introduction to two important concepts in this thesis: framework theories and proof editors. We mentioned in the introduction that framework theories (or framework logics) are formal systems designed to characterise other (object) theories. In the first section we give an overview of the these theories. Proof editors are computer systems where object theories can be represented and the system acts as a tool to develop proofs in the given theory. We examine these in the second section.

2.1 Framework Theories

Most framework theories are based on the *typed lambda calculus* [Church 40]. It is a suitable formalism for the development of framework theories because it handles variable substitutions in a uniform way through its two basic operations: abstraction and application. From a technical point of view, types provide a security system: type checking prevents incoherent operations from being attempted. But from a semantical point of view, types provide a natural link to logical systems through the Curry-Howard isomorphism [Howard 80] [Curry & Feys 58] in which propositions are mapped to types and type constructors are mapped to connectives. This isomorphism is exploited under the *propositions-as-types* paradigm in which types are propositions and their members are seen as proofs of the propositions.

The work on logical frameworks can immediately be traced back to the idea of a proof

checker: a system in which mathematical arguments could be represented and the machine could be used to verify them. A pioneering project along this line was the work of the *Automath* group in Eindhoven [Bruijn 80].

Started in 1966 the Automath project was the first project devised to manipulate mathematics with computers. The group developed a whole family of theories based on the typed lambda calculus [Church 40] and implemented them in the AUTOMATH system. This system was used to verify the Landau book of analysis [Benthem L.S. 77] in order to show the possibility of computer verified mathematics. Many of the concepts used later in newer theories and implementations were developed by the Automath group (e.g. name free lambda calculus) as well as being an inspirational source for many researchers.

Coquand and Huet's *calculus of constructions* is also based on type theory and provides a formalism to develop mathematics. Coquand and Huet show in [Coquand & Huet 85] how this theory can be used as a framework logic to represent many fragments of mathematical theories.

The Nuprl logic developed at Cornell [Constable *et al* 86] is based on Martin-Löf type theory [Martin-Löf 84]. Nuprl is a rich type theory designed to experiment with and develop proof techniques for a variety of mathematical theories and has been implemented in Nuprl proof editor. In [Basin & Constable 93] Nuprl logic is proposed as a framework theory.

In this thesis we focus on one particular framework theory, the Edinburgh Logical Framework.

2.1.1 The Edinburgh Logical Framework

The *Edinburgh Logical Framework* (LF) [Harper *et al* 92] is a formal system specifically designed as a framework theory. It is a typed lambda calculus with dependent function types which capture the notion of inference rule in a natural way. Object theories are represented in LF by specifying *signatures*. These are sets of typed identifiers which define term and formula constructors of the theory (e.g. connectives, operations, constant terms, etc.) and constants whose types represent the inference rules of the

object theory. It is particularly useful to represent logics in Natural Deduction and Hilbert styles [Avron *et al* 87].

Logics are represented in LF by exploiting the *judgements-as-types* paradigm whereby judgements are associated with the type of their proofs. This paradigm is an extension, proposed in [Harper *et al* 92], to *propositions-as-types* mentioned earlier.

LF has only one type constructor, Π. It is a dependent function type constructor abbreviated as $\rightarrow$ when the range is independent of the domain. Π is used to express universal quantification and implication.

This type constructor allows us to specify three kinds of judgement: atomic, hypothetical (e.g. $A \rightarrow B$ with A and B judgements) or schematic (e.g. $\Pi_{x:o}P(x)$ with o a type and P a judgement).

Judgements may represent theorems of the logic such as:

$$\Pi_{A:o}\Pi_{B:o}true(A \supset (A \supset B) \supset B) \tag{2.1}$$

where A and B are abstractions in type o which is defined to be the type of propositions. *true* is a judgement valued function from o into *Type*, the kind that contains all types. Schematic judgement 2.1 states that it is true that

$$A \supset (A \supset B) \supset B$$

for all propositions A and B. It can be read as an axiom schema. Judgements may also be used to represent inference rules like:

$$\Pi_{A:o}\Pi_{B:o}true(A) \rightarrow true(A \supset B) \rightarrow true(B)$$

which states that if A and $A \supset B$ are true, then B will also be true for all propositions A and B.

The usual format of inference rules is:

$$\Pi_{A_1:o} \cdots \Pi_{A_n:o}J_1(A_1, \ldots, A_n) \rightarrow \cdots \rightarrow J_m(A_1, \ldots, A_n) \rightarrow K(A_1, \ldots, A_n)$$

where J_i and K are judgement functions. We call Π_{A_i} the *quantification part* of the rule; we call $J_i(A_1, \ldots, A_n)$ the judgements in the *body* of the rule and $K(A_1, \ldots, A_n)$ the *head* of the rule.

In figure 2.1 there is an example of an LF signature.

$$
\begin{array}{rcl}
o & : & Type \\
true & : & o \rightarrow Type \\
\supset & : & o \rightarrow o \rightarrow o \\
\bot & : & o \\
 & & \\
\supset_i & : & \Pi_{A,B:o}(true(A) \rightarrow true(B)) \rightarrow true(A \supset B) \\
\supset_e & : & \Pi_{A,B:o}(true(A \supset B) \rightarrow true(A)) \rightarrow true(B)
\end{array}
$$

Figure 2.1: LF signature for Minimal Logic

Notice that inference rules depend on other type definitions in the signature. Given a signature for a specific logic, LF can be used as a meta theory to set up theorems of the logic and verify them by testing the corresponding judgement, as a type, for *inhabitance*. The LF system we use in this thesis is system L [Pym & Wallen 91]. The rules of this system are listed in Figure A.3 (Section A.2).

2.2 Proof Editors

Proof editors are systems where mathematical theories are represented by giving rules to form expressions, and rules of inference to reason about them. The user can then present a theorem in the theory and use the machine to do the symbolic manipulation required to prove the theorem. We now describe some proof editors in no particular order but later, in the summary, we present a table with their main characteristics.

LCF: Milner's group [Gordon *et al* 79] built a proof assistant called LCF to reason about computable functions (logic PP-λ). Their idea was to provide an environment with a meta-language (called ML) that enabled the user not only to prove theorems about computable functions but also construct programs in the meta-language to automate parts of the proof. These programs are what we referred to above as tactics and tacticals. LCF is the first system in which they were used.

Nuprl: The Nuprl system developed in Cornell [Constable *et al* 86] includes ideas from the previous systems. It is an Interactive Proof Development System for a type theory similar to Martin-Löf's Type Theory [Martin-Löf 79] that includes a library facility, abbreviations and tactic development facility in ML. Nuprl is perhaps, of all systems developed so far, the one with the richest type theory. Thus it has a strong expressive power but the basic typing relation is not decidable. Thus, proofs in Nuprl normally involve a large number of typing subgoals to be solved by the user. In [Constable & Howe 90] and [Basin & Constable 93] the use of Nuprl as a Framework Logic is explained.

Lego: LEGO [Luo & Pollack 92] is an implementation of various type systems:

- LF: Edinburgh Logical Framework [Harper *et al* 92]
- PCC: Pure Calculus of Constructions [Coquand & Huet 88].
- CC: (Generalised) Calculus of Constructions [Coquand 86].
- XCC: Extended Calculus of Constructions [Luo 89].

The user can specify in which system she wants LEGO to work. The system has been implemented in ML but, in contrast with LCF, there is no support for writing tactics and tacticals. Instead, a fixed set of commands is available to the user and no direct access to individual rules of any of the type systems mentioned above is allowed. Since the type systems are meant to be used as meta-logics, all commands in LEGO implement proof operations at the object level.

Oyster: Oyster is an implementation of Nuprl done by the Mathematical Reasoning group at Edinburgh. It is part of a system called Oyster-Clam [Bundy *et al* 90a]. Clam is a proof planner (described in Section 3.2) and Oyster is the proof editor where Clam's plans may be executed. Oyster has a library mechanism where theorems, definitions, tactics and other objects are stored. It uses Prolog as its tactic language. It has been used within the group to develop proof and synthesis techniques for theories like Peano arithmetic, functional programming, hardware specification and others.

Proofs in Oyster are constructed backwards. A proof is a tree whose root node contains the conjecture and whose leaves contain axioms. Each edge of the tree corresponds to an inference rule applied backwards.

Since tactics apply many inference rules, every time a tactic is applied, a section of the proof tree is expanded. A proof plan for a theorem in Oyster is also a tree. The root node is the tactics that must be applied to the conjecture and each node below is the tactic that must be applied to an output sequent of the parent tactic.

Mollusc: This proof editor [Richards *et al* 94], in contrast with most of the systems described here, does not implement a particular theory to be used as a generic meta-theory to represent object-level theories. Mollusc's design resembles more the idea of an *expert system shell* where the system has all the necessary tools to encode rules and concepts as well as a flexible inference engine and user interface for each user to customise his or her own particular expert system.

Mollusc is implemented in Prolog. It has a library mechanism to handle definitions, tactics, and other concepts definable by the user. It also has a language to specify the syntax of mathematical theories. Besides the syntax, the user must also specify the inference rules of the theory in question and how the theory's expressions are to be displayed on the screen.

Mollusc is a sequent-based backwards proof construction system. We have developed a version of LF in this system (Appendix A) and the system described in Chapter 8 generates proof plans for this implementation of LF.

Isabelle: Drawing from the experience of Cambridge LCF [Paulson 87], Paulson developed *Isabelle* [Paulson 94], a proof assistant based on higher-order logic. Although Isabelle's logic is not concerned with type inference (i.e. the rules do not express any typing properties), it does have a type hierarchy to classify the objects to be used. One of the consequences of this is that Isabelle does not build proof "objects" directly —like Nuprl or Lego— but is only concerned with the provability of propositions. Despite this, LF and Isabelle are very similar as far as provability is concerned [Paulson 90]. The range of logics representable in both is the same and the way to encode them

exploits the same principles except for the typing side of LF that makes its rules more complicated.

This proof editor is one of the most widely used to experiment with big proofs; many theories have been encoded in it. The system also provides the user with the facility of writing tactics and tacticals in ML to automate search.

λProlog and ELF: Felty has taken a different approach to logical frameworks by specifying theorem provers in a logic programming language [Felty 89]. This language includes simply-typed lambda terms, higher-order unification and an extension to Horn clauses (called higher order hereditary Harrop formulae) as basic expressions. She present different logics in the system as logic programs and theorems as goals to be proven using the inference mechanism of the language. Her system also provides an environment to write tactics and tacticals to guide search in the proofs. The language she uses is λ-Prolog [Miller & Nadathur 88].

Frank Pfenning has proposed a hybrid approach between type theory-based systems to present logics and logical programming [Pfenning 91]. His system Elf is a logic programming system that gives direct operational interpretation to LF connectives. In it, the extract term of the proof is automatically constructed through the search process.

These two approaches are proof editors with a uniform proof search mechanism, on top of which tactics may be programmed to make search more efficient.

Coq: This system is an implementation of the calculus of constructions extended to *Inductive Types*[Coquand & Paulin-Moring 88]. It is a tactic theorem prover with strong emphasis on program synthesis. The system has been programmed in CAML Light [Caml light] and synthesised programs are translated to CAML Light by Coq and can be readily executed.

Alf: Alf [Nordström 93] is an implementation of Martin Löf's monomorphic theory of types. The emphasis of this prover is on proof-object construction. Other provers focus on proving theorems represented by types while the system gradually constructs the

extract term; Alf focusses the user's attention on the proof object being constructed and helps her complete it by providing all the necessary typing information.

Alf is not a tactic theorem prover. It encourages the construction of theory corpus that can be used as lemmata in the current proof.

There is also a graphical version of this system where all expressions are displayed as they appear in mathematical textbooks and all commands are provided via menus on the screen accessible with a mouse. This interface makes for a good working environment.

HOL: HOL [Gordon 88], a proof editor based on Higher Order Logic, evolved from LCF into its present form. Proven theorems are stored in theory file to be used in further proofs. HOL has been designed to develop proof techniques through tactics and tactical written in its meta-language ML. There are a number of libraries already available that include support for the specification language Z and several process algebras such as CCS.

HOL is the system most widely used in industry with applications on safety critical systems and hardware verification.

2.3 Summary

In this chapter we have seen an overview of theories called framework logics and some implementations of proof editors. We described framework logics as meta-theories to be used to specify other theories called object theories. The purpose of framework logics is to account formally for operations common to many theories like function application, variable binding and substitution and others. Framework logics are then very useful to implement computer systems to prove theorems in various mathematical theories.

Some Proof Editors are implementations of specific theories and some are "generic" in the sense that they support the representation of theories by means of grammars and rule data bases. Some proof editors emphasize the incremental development of theories

by proving and using lemmata and some are designed as environments where the user develops proof techniques by programming tactics and tacticals. Some are based on type theory and stress the importance of extract terms while others are more focussed on provability.

Chapter 3

Proof Plans

Proof planning provide a general framework to develop and experiment with proof-automation techniques. The techniques presented in this thesis have been developed using proof plans and we use this chapter as an introduction to this framework. This chapter also describes *Clam*, an implementation of a proof planner, *rippling*, a proof guidance technique developed in proof plans and related topics which constitute the main vocabulary for the rest of the thesis.

3.1 Introduction

Proof plans is a technique where heuristics to guide the application of tactics are made explicit. The idea is to use a meta-logic to partially specify tactics and use a planning program to work out in advance what the proof is going to "look like" (i.e. make a plan). After a plan has been built for a theorem, it is used to construct the actual proof by applying the designated proof steps.

Specifications of tactics are called *methods* and consist of information describing, among other things, the circumstances under which the technique should be used (preconditions) and what the consequences of its application (i.e. what would be the new goals to prove). This information is used by a planner to chain methods to form a plan. The process works backwards; it begins by selecting a method applicable to the conjecture sequent and ends with one or more methods whose outputs are empty. Methods with empty outputs specify terminating tactics, that is, tactics that produce axioms.

Building proof plans has the advantage of not having to do all the operations needed to apply the tactic. The architecture of the methods and the meta-logic used in them enables a declarative specification of the heuristics for the tactics. This way, it is easy to edit and experiment with the heuristic information without altering the tactics themselves.

A proof plan does not guarantee a successful proof because it works on a "hypothetical" description of tactics, soundness is only ensured by the successful application of the plan in the proof editor.

3.2 Clam

Clam [vanHarmelen *et al* 93] is a proof planner for the Oyster proof editor (see Section 2.2). Clam builds proof plans from specifications of Oyster tactics. The specifications are called *methods* and are stored in a database accessible to a set of planning programs called *planners*. We now describe methods and planners.

3.2.1 The Methods

Clam has a library mechanism whereby methods and submethods may be loaded to be considered in the planning process. Submethods have the same syntax as methods but can only be called from within methods. This allows us to have a modular way of specifying tactics. Methods and submethods in Clam have the following slots:

- (sub)Method's name.
- Input formula.
- Preconditions.
- Postconditions.
- List of new goals.
- Tactic.

The name of the method goes into the plan; it may be a Prolog pattern containing variables to pass some parameters into the plan. The input formula is a Prolog pattern that acts as the first filter: it must match the current goal for the method to be applicable. If the input slot unifies with the current goal, the planner tries to satisfy the list of preconditions. If the preconditions succeed, the postconditions are tried; they serve the purpose of doing all the computation necessary to generate the list of new goals. The last slot contains a name or Prolog pattern representing the tactic call that must be made in the proof editor.

3.2.2 The planners

Clam has a series of planning programs each one corresponding to a particular search strategy: depth-first planner, breadth-first planner, depth-first iterative-deepening planner and best first planner. The first three are brute force planners; they select the methods in the order in which they appear in the method data base. This order has a big effect in the planning process because many methods can be applicable to the same sequent. The best-first planner, however, has a general heuristic function to order the methods before they are considered can be defined.

Clam also has a hint mechanism [Negrete 91] and [Negrete 93]. This mechanism allows the user to provide the planner with hints before its starts planning. Hints are instructions to alter the planning process in specific cases. Using this facility, the user can alter the order in which the methods are used in order to reduce the search at certain stages.

3.3 Rippling

Rippling is a technique to guide the application of rewriting rules when we know beforehand the shape of the final formula of a rewriting process. When certain normal forms exist for a formula, it is sometimes possible to identify what substructures of it need to be changed or eliminated in order to obtain the desired form. If these substructures are identified, the rewriting process can be guided by applying only rewrites that change those structures.

The idea of rippling originated for inductive proofs [Bundy *et al* 88]. It couples with another technique called *recursion analysis*, originally built in the Boyer/Moore theorem prover ([Boyer & Moore 79]). Recursion analysis consists of analysing a conjecture in order to select an induction scheme as well as an induction variable to apply induction to. Both techniques have been implemented as part of a Clam method.

When proving theorems by induction, we have, in the inductive step, an induction conclusion and one or more induction hypotheses. The proof normally involves manipulating the induction conclusion until an induction hypothesis can be used to reduce it. This reduction is called *fertilisation*. We can think of the induction hypothesis used as the target for the process of rewriting the conclusion.

The induction conclusion and hypothesis are usually very similar. The differences are constructor functions applied to the variable where the induction is taking place. If these constructor functions are removed from the induction conclusion, we will have an exact copy of an induction hypothesis. We can use this information to guide a rewriting process to transform the induction conclusion into the required form.

Let's see an example. Using axioms for Peano arithmetic, we prove the associativity of addition:

$$\Pi_{x,y,z:pnat}.(x + y) + z = x + (y + z)$$

where *pnat* is the type of Peano numbers. From the definition of addition, the following rewrites are obtained:

$$0 + X \Longrightarrow X \tag{3.1}$$

$$s(X) + Y \Longrightarrow s(X + Y) \tag{3.2}$$

and the following rule is obtained from the definition of equality (applied backwards):

$$s(X) = s(Y) \Longrightarrow X = Y \tag{3.3}$$

The first step is induction on x. The base case is:

$$(0 + y') + z' = 0 + (y' + z')$$

where x',y',z',etc. represent fresh variables. This case is solved by applying elementary rewrite rules like 3.1. The interesting part of the proof, at least as far as rippling is concerned is the step case. We mark the differences between the induction hypothesis and conclusion as follows:

Induction hypothesis: $(x' + y') + z' = x' + (y' + z')$

Induction conclusion: $(\boxed{s(\underline{x'})}^{\uparrow} + y') + z' = \boxed{s(\underline{x'})}^{\uparrow} + (y' + z')$

The boxes indicate the structures that have to be removed and the underlined variables in them remind us that these are going to be preserved, only the functors have to be removed. The term we use for the partial term obtained by removing the underlined term from the expression in the box is *wave front*. The underlined expressions are called *wave holes*. The expression obtained after removing the wave fronts is called the *skeleton* and in this case is equal to induction hypothesis. The term rippling comes from the effect of rippling the wave front out of the way. The arrows on the boxes indicate the direction in which the wave fronts must be rippled. In this case, the wave fronts have to be *rippled out*, that is, up the term tree representing the formula.

To guide the simplification of the goal, rewrite rules are annotated to record the movement of the wave fronts when they are applied. These rewrite rules with annotations are called *wave rules*. The wave rule used in this example is the following obtained from rewrite 3.2:

$$\boxed{s(\underline{X})}^{\uparrow} + Y \Longrightarrow \boxed{s(\underline{X + Y})}^{\uparrow} \tag{3.4}$$

Notice how the wave front moves out in the term when the rule is applied. Wave rules are applicable to an annotated term when the left-hand side of the rule matches the term including the annotations. This restriction reduces the number of applicable rules to those which ripple the annotations on the term. This usually reduces the number of applicable rewrites to only a few.

In our example, applying rule 3.4 to both sides of the induction conclusion, produces:

$$\boxed{s(\underline{x' + y'})}^{\uparrow} + z' = \boxed{s(\underline{x' + (y' + z')})}^{\uparrow}$$

Using the same wave rule again, gives:

$$\boxed{s(\underline{(x' + y') + z'})}^{\uparrow} = \boxed{s(\underline{x' + (y' + z')})}^{\uparrow}$$

At this point, the expression is said to be *fully rippled* because no more rippling is possible and the wave fronts are at the out-most position (we say they are *beached*). In some cases, wave fronts are not beached but disappear (we say they *peter out*). When no more rippling is possible and the wave fronts are beached or peter out we also say that the expression is fully rippled. There is also the case when no more rippling is possible but the wave fronts have neither been beached nor petered-out. In such cases we say that rippling is *blocked.*

The next step in the example proof is to simplify the expression using the following wave rule obtained from the definition of equality:

$$\boxed{s(\underline{U})}^{\uparrow} = \boxed{s(\underline{V})}^{\uparrow} \Rightarrow U = V$$

to obtain:

$$(x + y) + z = x + (y + z)$$

which is identical to the induction hypothesis and we can now fertilise and finish the proof.

As we said before, the arrows on the boxes indicate the direction of rippling. There are other directions for rippling but in this thesis we will only use the outwards type. Therefore, from now on, we won't write arrows on the boxes.

In general, wave holes may contain annotated terms as well to form more complex annotated terms:

$$true(w_1, \boxed{\Box \boxed{a \supset \underline{b}}})$$

the skeleton is: $true(w_1, b)$, the wave fronts are $\Box$ and $a \supset$. There are two wave holes: $\boxed{a \supset \underline{b}}$ and b. Sometimes we "merge" wave fronts to obtain a more readable form. The *merged form* of the example above is:

$$true(w_1, \boxed{\Box(a \supset \underline{b})})$$

Merging consists of removing wave fronts which are immediately contained in a hole together with the hole itself. This procedure does not affect the skeleton defined by

the annotations. The removed wave fronts and holes do not contribute to the skeleton because they cancel one another. These annotations are not superfluous however, intermediate wave fronts can be shifted individually by rewrite-rule applications, without the whole expression having to change. We refer to the non-merged form as *split form* and will make use of them both throughout the thesis as needed.

One of the main properties of wave rules is that the skeleton on both sides of them is the same, that is, they are *skeleton preserving.* Rippling is terminating. Rewrite rules are made into wave rules only when annotation on the right-hand-side of the rule is closer to the end point than the one on the left-hand-side. A measure for this and a proof of termination can be found in [Basin & Walsh 94].

Clam uses a restricted version of an algorithm called *Difference Matching* [Basin & Walsh 91] to annotate the induction conclusion. This algorithm matches a term and a pattern by structure hiding as well as variable substitution. The result of difference matching a pattern and a target expression is a substitution of variables, as in normal matching, and an annotated target whose skeleton matches the pattern with the substitution.

3.3.1 Coloured Rippling

There are cases, especially in inductive proofs, where there is more than one hypothesis to ripple the goal towards. Each hypothesis has its own difference matchable subexpression in the goal and what is needed to prove the theorem is to ripple the different sections of the goal simultaneously towards the corresponding hypotheses.

In these cases we consider the goal to have many skeletons, one for each of the mentioned hypotheses; we distinguish the skeletons with colours. This way, the rippling process may be controlled by keeping track of the skeletons of the hypotheses separately. The rules that ripple more than one colour are called *coloured wave rules.* Using this kind of rule and coloured annotations, we prevent the rippling process from mixing skeletons that correspond to different hypotheses. This type of rippling is called *coloured rippling* and is described in [Yoshida *et al* 94]; here we just give an example from that paper: Consider the definition of the function *maxht* (maximum height) over the

recursive type of binary trees with constructors: *node* and *leaf*:

$$\begin{aligned} maxht(leaf(n)) &= 0 \\ maxht(node(t_1, t_2)) &= s(max(maxht(t_1), maxht(t_2))) \end{aligned}$$

We can similarly define *minht* (minimum height) and form the following conjecture:

$$\forall t : tree(pnat).maxht(t) \geq minht(t)$$

A proof attempt by structural induction would have two hypotheses:

$$maxht(l) \geq minht(l)$$

$$maxht(r) \geq minht(r)$$

and an annotated induction conclusion:

$$maxht(\boxed{node(\underline{l}, \underline{r})}) \geq minht(\boxed{node(\underline{l}, \underline{r})})$$

Here we have both l and r in the skeleton. The first one should be rippled to match the first hypothesis while the second one should match the second hypothesis. The problem we have, if we consider all of these variables to be part of the same skeleton is that they may end up in the wrong place after Rippling. Consider the following wave rules:

$$maxht(\boxed{node(\underline{L}, \underline{R})}^{\uparrow}) \Rightarrow \boxed{s(max(\underline{maxht(L)}, \underline{maxht(R)}))}^{\uparrow} \quad (3.5)$$

$$minht(\boxed{node(\underline{L}, \underline{R})}^{\uparrow}) \Rightarrow \boxed{s(min(\underline{minht(L)}, \underline{minht(R)}))}^{\uparrow} \quad (3.6)$$

$$\boxed{s(\underline{X})}^{\uparrow} \geq \boxed{s(\underline{Y})}^{\uparrow} \Rightarrow X \geq Y \quad (3.7)$$

$$\boxed{max(\underline{U_1}, \underline{U_2})}^{\uparrow} \geq \boxed{min(\underline{V_1}, \underline{V_2})}^{\uparrow} \Rightarrow \boxed{\underline{U_1 \geq V_1} \wedge \underline{U_2 \geq V_2}}^{\uparrow} \quad (3.8)$$

$$\boxed{max(\underline{U_1}, \underline{U_2})}^{\uparrow} \geq \boxed{min(\underline{V_1}, \underline{V_2})}^{\uparrow} \Rightarrow \boxed{\underline{U_1 \geq V_2} \wedge \underline{U_2 \geq V_1}}^{\uparrow} \quad (3.9)$$

Rippling with wave rules 3.5,3.6 and 3.7 leads us to:

$$\boxed{max(\underline{maxht(l)}, \underline{maxht(r)})}^{\uparrow} \geq \boxed{min(\underline{minht(l)}, \underline{minht(r)})}^{\uparrow}$$

and now, if we use wave rule 3.8, we obtain the conjunction of the two induction hypotheses and the proof is finished. However, if we use wave rule 3.9 instead, we get

l and r in the wrong place:

$$\boxed{\underline{maxht(l) \geq minht(r)} \wedge \underline{maxht(r) \geq minht(l)}}^{\uparrow}$$

Annotations are intended to constrain the application of rewrite rules to those which help isolate the appropriate skeleton. We cannot rely however on the order of the rules to select the right wave rule. What we need is to further constrain the applicability so that the wrong rules are not selected. We notice that the annotations on the goal correspond to matching with two hypotheses. Rippling, in order to profit from the rewriting, must isolate the skeletons corresponding to each hypothesis. It is possible to do this by keeping track of each skeleton separately. We do this by colouring wave holes to distinguish different skeletons. Each rule must preserve skeletons of every colour.

If from the beginning we consider that the two hypotheses match different skeletons in the goal, we would obtain an annotated goal like the following:

$$maxht(\boxed{node(\underline{l}_{c_1}, \underline{r}_{c_2})}^{\uparrow}) \geq minht(\boxed{node(\underline{l}_{c_1}, \underline{r}_{c_2})}^{\uparrow})$$

Wave rules become coloured wave rules:

$$maxht(\boxed{node(\underline{L}_{C_1}, \underline{R}_{C_2})}^{\uparrow}) \Rightarrow \boxed{s(\max(\underline{maxht(L)}_{C_1}, \underline{maxht(R)}_{C_2}))}^{\uparrow} \tag{3.10}$$

$$minht(\boxed{node(\underline{L}_{C_1}, \underline{R}_{C_2})}^{\uparrow}) \Rightarrow \boxed{s(\max(\underline{minht(L)}_{C_1}, \underline{minht(R)}_{C_2}))}^{\uparrow} \tag{3.11}$$

$$\boxed{s(\underline{X}_{C_1})}^{\uparrow} \geq \boxed{s(\underline{Y}_{C_1})}^{\uparrow} \Rightarrow X \geq Y \tag{3.12}$$

$$\boxed{max(\underline{U_1}_{C_1}, \underline{U_2}_{C_2})}^{\uparrow} \geq \boxed{min(\underline{V_1}_{C_1}, \underline{V_2}_{C_2})}^{\uparrow} \Rightarrow \boxed{\underline{U_1 \geq V_1}_{C_1} \wedge \underline{U_2 \geq V_2}_{C_2}}^{\uparrow} \tag{3.13}$$

$$\boxed{max(\underline{U_1}_{C_1}, \underline{U_2}_{C_2})}^{\uparrow} \geq \boxed{min(\underline{V_1}_{C_2}, \underline{V_2}_{C_1})}^{\uparrow} \Rightarrow \boxed{\underline{U_1 \geq V_2}_{C_1} \wedge \underline{U_2 \geq V_1}_{C_2}}^{\uparrow} \tag{3.14}$$

Rippling now proceeds as follows:

$$\boxed{max(\underline{maxht(l)}_{c_1}, \underline{maxht(r)}_{c_2})}^{\uparrow} \geq \boxed{min(\underline{minht(l)}_{c_1}, \underline{minht(r)}_{c_2})}^{\uparrow}$$

$$\boxed{\underline{maxht(l) \geq minht(l)}_{c_1} \wedge \underline{maxht(r) \geq minht(r)}_{c_2}}$$

leading us to the desired goal. The difference with the monochromatic case is that coloured wave rule 3.14 above is not applicable.

The importance of describing coloured rippling is that we shall be using it in our methods to keep different proof paths apart and prevent the disappearance of connections under the application of some rewrite rules. More on this in Chapter 8.

3.4 Summary

In this chapter we motivated and described proof plans as a paradigm to automate proof search and experiment with different proof-automation techniques. We also introduced the Clam proof planner and described its planners and methods.

Rippling, a technique to guide the application of rewrite rules, was motivated and its main concepts introduced. We mentioned coloured rippling as an extension to the original rippling. In Chapter 6 a formalisation of coloured annotated terms will be introduced. This formalisation is an extension of the concepts introduced informally in this chapter.

Rippling, although originally developed for inductive proofs, has evolved to become a general technique of its own as part of *difference reduction* [Basin & Walsh 96]. In the next chapter we give an overview of how we use difference reduction to build proof plans for framework logics.

Chapter 4

Proof Planning with Logic Presentations

In this chapter we present an overview of the work described in this thesis, its various parts, how they relate to each other and a simple illustrative example. We do this in order to prepare the reader for the following chapters, where the work is presented in detail, by giving an overall understanding of how all the ideas and algorithms fit together.

4.1 Introduction

This thesis is about proof search; we develop search techniques that can be used to automate the proof process for several logics. We are not interested in giving a fixed list of techniques for a particular logic. What we present here is a uniform way of interpreting Natural Deduction presentations of logics to develop proof automation techniques as methods in the framework of proof plans.

When doing search in framework logics, we often find that we want to produce *connections* between hypotheses and conclusions (i.e. identical expressions on both sides of the sequent) to obtain axioms. The word connection here is taken by analogy to the Connection Method [Bibel 82]. In this method, complementary formulae (connection) are identified in a matrix-based description of the conjecture. Theoremhood there is defined in terms of paths of connecting formulae through the matrix. In order to obtain such *complementary* expressions in our work, we first look for them as subexpressions

of the current hypotheses and goals at some step of a proof. Once we have located the two expressions that might make a connection, we start applying inference rules that isolate the desired formulae in the appropriate side of the sequent. Connecting formulae are identified by a polarity value assigned to each expression. Polarity values can be $+,-$ and $\pm$ and are assigned by an algorithm described in Chapter 5. Only unifying expressions with different polarity values constitute a potential connection.

We present a setup whereby using proof plans we can reason about proofs in a framework logic by interpreting inference rules as rewrite rules and using rippling and other techniques to automate the process of isolating connecting expressions. This approach is based on the principle of *difference reduction* mentioned in Section 1.4.2.

Difference reduction provides a pattern to develop different search strategies. By alternating difference unification and control techniques (such as rippling) to reduce differences, we can build general methods to plan proofs in Logical Frameworks. A Proof Plan in our setting will be composed mainly of methods that apply various difference reducing techniques based on rewriting. These methods will use rewrite rules extracted from the inference rules of the particular logic internalised in the framework.

4.2 General Description

As mentioned in Section 1.4.2, we use techniques similar to those used in inductive theorem proving to guide search in framework logics. There are important differences between the two approaches however:

1. Since in our setting we don't have a fixed induction hypothesis, fertilisation could potentially be achieved with any hypothesis. In fact, it is likely that the appropriate hypothesis will need to be constructed, as opposed to having it from the start.

2. In the inductive case, we difference match the goal with the induction hypothesis and annotate only the former to indicate the difference. After this we ripple the goal. In our case we may also need to apply rewrite rules to hypotheses. We can rewrite hypotheses to match goals, rewrite goals to match hypotheses or rewrite both to a common expression. This means we need to annotate hypotheses as well as goals and therefore we need to use difference unification rather than difference matching (Section 4.2.2).

3. Refining goals with inference rules replaces the goal with a new goal in the new proof state. This can be seen as rewriting the goal with the inference rule treated as rewrite rule[1]. Forward-chaining a rule with a hypothesis, however, adds a new hypothesis, it does not rewrite it. To maintain the symmetry with the goal case, we also call this operation rewriting in spite of this fact.

Bearing all the previous considerations in mind, we may now set out to describe difference reduction techniques to guide proof search in a framework logic. The proof techniques are implemented as methods to be used by a planner in the construction of a proof plan for the theorem. The application of the plan, in the particular implementation of the framework logic, will consist of applying tactics parameterised by the object logic that corresponds to the theorem.

The inference rules of a framework logic are used to handle deductions of the inference rules of the object logic. Inference rule applications, at the framework level, don't usually correspond to any particular describable proof step at the object level. To achieve a full proof step at the object level we normally require the application of several inference rules of the framework logic. This property is related to the topic of granularity mentioned in Chapter 1. Inference rule applications at the object logic translate into finer steps at the framework level. Doing a refinement step for the object logic, for instance, requires unification and stepwise instantiation of each of the universally quantified variables in the (object-level) inference rule. Each of these instantiations corresponds to an inference rule application of the framework logic. All these framework level inference rule applications that correspond to one object level rule application are assumed to have been programmed as tactics in a proof editor.

Unless stated otherwise, we will use the term inference rule to refer to the object level logic's inference rules. The application of the framework's inference rules will be transparent to us —and to the user of the system— and we will only refer to higher level operations on the framework logic as defined by the tactics.

Our approach to proof planning for logical presentations is divided into four stages:

[1] It may also be the case that a refinement produces more than one new proof state; the rewrite rules to deal with this case will be described in Section 4.2.3

1. Balancing.
2. Comparison.
3. Rippling.
4. Fertilisation.
5. Unblocking.

Balancing When a proof is begun, normally there are no hypotheses. They appear as the proof proceeds through applications of introduction rules. In order to be able to obtain connections across the sequent symbol through Rippling, we need to justify the number of connections before starting the rewriting process. This is achieved by combining the application of introduction rules from the object logic and the framework logic to the conjecture. The object logic introduction rules cause the goal to fragment into smaller judgements linked by framework connectives and the framework rules introduce the new judgements into the hypothesis list. The process continues until a maximum number of potential connections is reached. This maximisation of potential connections is achieved in the first stage mentioned above. We will see an example of it below in Section 4.3.

Comparison The second stage consists of difference unifying the goal and the hypotheses and ordering the set of annotated goal-hypothesis pairs. The order given to the set is induced by a measure of the difference between the members of the pair. This way, the members of the set of pairs will be selected in order.

Rippling The third stage consists of rippling both the goal and the selected hypothesis using wave rules extracted from the signature. Each time a wave rule is applied to the hypothesis the rewriting of it is reflected as a new hypothesis. The annotations are only kept on the last hypothesis so that a new rule may be applied to it.

Fertilisation The fourth stage consists of fertilising, that is, making a connection. The process consists of identifying connecting expressions in the sequent and reducing

the sequent by making the connection. Fertilisation is usually possible after a successful rippling run. We have two ways of fertilising: backwards and forwards.

If one expression is the goal or is the head of the goal and the corresponding connecting expressions in the hypothesis is a hypothesis or the head of a hypothesis, then the connection can be made by backward-chaining the goal and the corresponding hypothesis.

For example, if the connection in the context is a hypothesis on its own, the sequent is trivial:

$$\cdots, j \vdash k_1 \rightarrow \cdots \rightarrow j$$

If the hypothesis containing the connection is a conditional judgement, then the hypothesis is used backwards as a derived inference rule to make the connection. We go from:

$$\cdots, l_1 \rightarrow \cdots \rightarrow j \vdash k_1 \rightarrow \cdots \rightarrow j$$

into:

$$\begin{array}{lll} \cdots, l_1 \rightarrow \cdots \rightarrow j, k_1, \cdots, k_n & \vdash & l_1 \\ & & \vdots \\ \cdots, l_1 \rightarrow \cdots \rightarrow j, k_1, \cdots, k_n & \vdash & l_n \end{array}$$

If one of the expressions is part of the body of a hypothesis and the complementary expression is a hypothesis, then the connection is made by forward chaining. We go from:

$$\begin{array}{lll} & \vdots & \\ hyp_1 & : & j \\ hyp_2 & : & l_1 \rightarrow \cdots j \rightarrow l_n \\ & \vdash & k \end{array}$$

to:

$$\begin{array}{lll} & \vdots & \\ hyp_1 & : & j \end{array}$$

$$
\begin{aligned}
hyp_2 \quad &: \quad l_1 \rightarrow \cdots \rightarrow j \rightarrow \cdots l_n \\
hyp_3 \quad &: \quad l_1 \rightarrow \cdots \rightarrow l_n \\
&\vdash \quad k
\end{aligned}
$$

After fertilisation, the branch of the plan is either complete or there are new sequents to solve. In the latter case the whole process is repeated.

Unblocking The system's strategy is to first reduce differences with wave rules because it is the most constrained way of reasoning about inference rule application. Not all rewrite rules parse into wave rules however. For this reason, if the application of wave rules fails, the unblocking stage tries to apply rewrite rules to unblock the rewriting process and go back to rippling. As before, not all inference rules translate into rewrite rules so, if also rewrite rule application fails, unblock tries the direct application of inference rules.

4.2.1 Polarity

At the beginning of this chapter, we mentioned that connecting expressions are identified by their polarities. The assignment of polarity also plays a role in the selection of wave rules.

Inference rules may be used to refine the goal or may be applied forward to hypotheses. For this reason, from each inference rule two rewrite rules may be extracted: one that corresponds to the application of the inference rule left-to-right (forward-chain) and one right-to-left (refine).

The assignment of polarity values to subexpressions depends on the particular logic a formula corresponds to. In the original version of rippling, the computation of polarity values is done with respect to Oyster's logic and the procedure to do it is hardwired in the system to be computed whenever an annotated term is to be rippled. In this thesis all framework level polarity is computed with respect to LF.

We require wave rules to rewrite subexpressions of goals or hypothesis. For this reason, we need to be able to foresee which side of the sequent a subexpression potentially

belongs to in order to know what wave rules are applicable to it. The simplest cases are the constructors of the framework logic. In LF, a goal is assigned a positive polarity; we mark this with a positive sign as a superscript as follows:

$$\vdash (j_1^- \rightarrow j_2^+)^+$$

Inference rules are applied to goals right-to-left so, rewrite rules obtained from them will have positive polarity to match a goal. Since the constructor $\rightarrow$ can be introduced in such a goal leaving j_1 to the left of $\vdash$ and therefore inference rules are applicable to it left-to-right, j_1 is assigned negative polarity. When used as a hypothesis, the polarities are reversed:

$$(j_1^+ \rightarrow j_2^-)^- \vdash G$$

The cases for constructor Π are similar:

$$\vdash \Pi_{x:o^-} P(x)^+$$

and

$$\Pi_{x:o^-} P(x)^+ \vdash G$$

The polarity values for o are added for completeness but are not used in practice, so we will not write them from now on. This way, an expression whose polarity is negative can be rewritten by a wave rule that encodes a forward-chaining operation. Conversely, positive polarity allows rewriting that encodes backward-chaining.

In this thesis, we take one more step forward and not only annotate expressions at the framework level —as described above— but also annotate expressions at the object level. This way, we can make a finer analysis to find connections at the object level. The potential connections can be simplified by rewriting until they become judgements of their own and are suitable for fertilisation as described in Section 4.2.

The assignment of polarity values depends on the encoding of the logic in the framework logic; different logics lead to different polarity assignments. We have developed an algorithm to assign polarity values to subexpressions of a formula with respect to a particular signature. Since the computation of these values may be computationally expensive, we do the assignment of polarity at the *comparison* stage explained in Section 4.2. When wave rules are extracted from the signature, they are also annotated

to record the transition of polarities through rewriting. This means that rippling spreads both wave and polarity annotations through the planning process.

Our polarity algorithm assigns three polarity values to object level expressions: $+$,$-$ and $\pm$. The first two correspond to the ones described above for the framework level. The value $\pm$ means that the algorithm could not assign a definite value to the subexpression, either because the signature does not provide for it or because it is ambiguous. Two polarity values are *compatible* if they are not either two pluses or two minuses. See Chapter 5 for a more detailed discussion of polarity.

Terms with polarity annotations are said to be *polarised.* Polarity values at the object level are used by the PCDU algorithm (described in the following section) to identify potential connections between two polarised terms. Two polarised terms with this property are said to be *compatible modulo polarity.* See Chapter 6 for a more detailed discussion of this topic.

4.2.2 Polarised Coloured Difference Unification

Difference unification is an algorithm similar to difference matching described in section 3.3. It uses the same kind of annotations but lays them on both terms being compared (see [Basin & Walsh 93] for details). Two terms are unifiable under this algorithm (*d-unifiable* for short) if the skeletons of the two terms are unifiable.

The algorithm we use in this thesis is called *polarised coloured difference unification* (PCDU). It differs from Difference Unification in that:

1. It uses a different formulation of annotated terms. The main characteristic of this formulation is that terms have polarity values.
2. Annotations are coloured. The algorithm computes all possible simultaneous skeletons each term can have and assigns different colours to them as in coloured rippling (Section 3.3.1).
3. It is ground. The terms to be compared have no variables.

Two terms are d-unifiable under PCDU if they can be annotated in such a way that the set of skeletons of both terms are compatible modulo polarity. A more detailed

description of PCDU can be found in Chapter 6. Since we don't use difference unification in this thesis, every time we write "d-unifiable" we refer to difference unification under PCDU.

4.2.3 Analysing Logic Presentations

Logics can be internalised in framework logics by defining the signature of a logic with the various constructors of the framework. In our system, we analyse logic presentations to extract rewrite rules and wave rules from them as was mentioned earlier.

The description of the procedure to extract rewrite and wave rules from signatures is given in Chapter 7; here we give a brief explanation and some example rules that will be used in Section 4.3 below.

In a signature for propositional logic in LF [Avron *et al* 87] we find the following rules:

$$\begin{array}{rcl}
\supset_i & : & \Pi_{A,B:o}(true(A) \rightarrow true(B)) \rightarrow true(A \supset B) \\
\supset_e & : & \Pi_{A,B:o} true(A \supset B) \rightarrow true(A) \rightarrow true(B) \\
\wedge_i & : & \Pi_{A,B:o} true(A) \rightarrow true(B) \rightarrow true(A \wedge B) \\
\wedge_{el} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(A) \\
\wedge_{er} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(B) \\
\vee_{il} & : & \Pi_{A,B:o} true(A) \rightarrow true(A \vee B) \\
\vee_{ir} & : & \Pi_{A,B:o} true(B) \rightarrow true(A \vee B) \\
\vee_e & : & \Pi_{A,B:o} true(A \vee B) \rightarrow \Pi_{C:o}((true(A) \rightarrow true(C)) \\
 & & \rightarrow (true(B) \rightarrow true(C)) \rightarrow true(C)) \\
abs & : & \Pi_{A:o} true(\bot) \rightarrow true(A)
\end{array} \tag{4.1}$$

Roughly[2], the rewrite rules are extracted from the signature as follows:

1. For each inference rule, add rewrite rules corresponding to all the possible ways the inference rule can be applied forwards (called left-to-right rule or *lr-rule*); this process may produce *twin-rules* (see below).

2. Discard the rules whose left hand side is *unconstrained.*

3. Simplify the remaining rules where possible.

4. Assigning positive polarity to both sides of rl-rules and negative polarity to both sides of lr-rules.

[2] A more precise definition of this process will be given in Chapter 7.

5. Polarise (i.e. add polarity annotations) both sides of the rules.

Unconstrained rewrite rules are those whose left-hand-side is applicable to any or almost any expression. One of the usual constraints for rewrite rules in the literature is that their left hand sides are not variables. In the method we use to extract rewrites, there will never be variables in the left-hand sides of the rules. However, we still put a constraint on rewrite rules to avoid rewrites that are practically unconstrained. Those rules are the rules whose judgement in the left-hand side has a variable as argument. For example, after step 1 above we obtain rules like:

$$true(A) \Longrightarrow (true(A \supset B) \rightarrow true(B))$$

We avoid this kind of rule because they are too unconstrained.

The simplification of rules in Step 3 consists of transforming the rewrites obtained in the previous steps. This simplification step uses predetermined procedures to obtain optimised versions of rules that are more suitable for rippling. In the next section, when we introduce *non-standard rules*, we will see an example of such an optimisation for the $\vee_e$ rule.

From the inference rules listed above we obtain the following set of wave rules[3]:

$$true(A^- \supset B^+)^+ \Longrightarrow (true(A)^- \rightarrow true(B)^+)^+ \qquad (rw{-}\supset_i)$$

$$true(A^+ \supset B^-)^- \Longrightarrow (true(A)^+ \rightarrow true(B)^-)^- \qquad (rw{-}\supset_e)$$

$$true(A^- \wedge B^-)^- \Longrightarrow true(A)^- \qquad (rw{-}\wedge_{el})$$

$$true(A^- \wedge B^-)^- \Longrightarrow true(B)^- \qquad (rw{-}\wedge_{er})$$

$$true(A^+ \vee B^+)^+ \Longrightarrow true(A)^+ \qquad (rw{-}\vee_{il})$$

[3] It is also possible to obtain rewrite rules from lemmata proved by the user. These could also produce useful wave rules. For example:

$$true(a \supset b) \rightarrow true(b \supset c) \rightarrow true(a \supset c)$$

produces twin wave rule:

$$true(\boxed{\underline{a^-}_{\{c_1\}} \supset \underline{c^+}_{\{c_2\}}})^+ \Longrightarrow \begin{cases} true(\boxed{\underline{a^-}_{\{c_1\}} \supset b^+})^+ \\ true(\boxed{\underline{b^-}_{\{c_2\}} \supset c^+})^+ \end{cases} \qquad (wr{-}trans)$$

$$true(A^+ \vee B^+)^+ \Longrightarrow true(B)^+ \qquad (rw{-}\vee_{ir})$$

$$true(\bot)^- \Longrightarrow true(A)^- \qquad (rw{-}\bot)$$

They correspond to $\supset_i$, $\supset_e$, $\wedge_{el}$, $\wedge_{er}$,$\vee_{il}$,$\vee_{ir}$ and $\bot_e$ in that order. The versions of these rules in the opposite direction are unconstrained so they are discarded. Rules $\wedge_i$ and $\vee_e$ also produce rewrite rules but they are non-standard. We discuss these in the next section.

From the rewrite rules obtained we can now obtain wave rules as follows:

1. Use Polarised Coloured Difference Unification to annotate both sides of the rewrite.
2. Discard those rules which are not measure decreasing.

Following these steps we obtain from the rewrites above the following wave rules:

$$true(\boxed{\underline{A^-}_{C_1} \supset \underline{B^+}_{C_2}}_{C_3})^+ \Longrightarrow \boxed{\underline{true(A)^-}_{C_1} \rightarrow \underline{true(B)^+}_{C_2}}^{+}_{C_3} \qquad (wr{-}\supset_i)$$

$$true(\boxed{\underline{A^+}_{C_1} \supset \underline{B^-}_{C_2}}_{C_3})^- \Longrightarrow \boxed{\underline{true(A)^+}_{C_1} \rightarrow \underline{true(B)^-}_{C_2}}^{-}_{C_3} \qquad (wr{-}\supset_e)$$

$$true(\boxed{\underline{A^-}_{C} \wedge B^-}_{C})^- \Longrightarrow \underline{true(A)^-}_{C} \qquad (wr{-}\wedge_{el})$$

$$true(\boxed{A^- \wedge \underline{B^-}_{C}}_{C})^- \Longrightarrow \underline{true(B)^-}_{C} \qquad (wr{-}\vee_{ir})$$

$$true(\boxed{\underline{A^+}_{C} \vee B^+}_{C})^+ \Longrightarrow \underline{true(A)^+}_{C} \qquad (wr{-}\vee_{il})$$

$$true(\boxed{A^+ \vee \underline{B^+}_{C}}_{C})^+ \Longrightarrow \underline{true(B)^+}_{C} \qquad (wr{-}\bot)$$

The rule **rw-$\bot$** cannot be converted into a wave rule because its two sides are not d-unifiable.

Weakening Coloured Wave Rules

The rules above are all coloured wave rules. They are used to ripple one or more colours (skeletons) at the same time. When two wave rules are equal as rewrite rules but the

set of skeletons of one of them is a subset of the set of skeletons of the other one, we say that the wave rule with less fewer skeletons is a *weakened* version (or a *weakening*) of the other one. There are cases where weaker versions of the wave rules originally computed from a signature are needed. These can be obtained by removing annotation corresponding to some skeletons from the wave rules as needed, with the condition that at least one skeleton remains. For example, a weakening of wave rule **wr-**$\supset_i$ is:

$$true(\boxed{\underline{A^-_{\{C_1\}} \supset B^+}}_{\{C_1\}})^+ \Longrightarrow \boxed{\underline{true(A)^-_{\{C_1\}} \rightarrow true(B)^+}}^+_{\{C_1\}}$$

Non-Standard Rules

We use some special kinds of rewrite rules which are different from the usual definition of rewrite rules. In this section we describe the characteristics which make them non-standard. We will only talk about rewrite rules but the same concepts extend to wave rules. Also, one rule can have more than one or all of the following non-standard characteristics.

Improper Rewrite Rules The first non standard rewrite rule that appears already in the list above, is rewrite rule **rw-**$\bot$. This rule has a variable in the right-hand side which does not appear in the left hand side. We call this kind of rule a *improper* rewrite rule (c.f. [Klop 92]). These rules introduce meta-variables in the proofs whose instantiation has to be deferred. We will consider the consequences of this in Section 4.4 and will see examples of how improper rules are used in Chapter 9.

Twin Rewrite Rules Rule $\wedge_i$, when interpreted right-to-left in Step 1, is transformed into the rewrite rule:

$$(true(B) \rightarrow true(A \wedge B)) \Longrightarrow true(A)$$

This rule does not convey the meaning of the introduction of a conjunction, that is, "to prove a conjunction, it is necessary to prove each conjunct". We simplify this kind of rule in step 3 by creating a *twin-rule* (*twin rewrite rule*, *twin wave rule*). Twin-rules are non-standard rewrite rules that rewrite an expression in two different ways. The

two ways are reflected in two copies of the original expression. We represent them using a key. For example, the twin rewrite rule corresponding to $\wedge_i$ is:

$$true(\boxed{\underline{A}^+_{C_1} \wedge \underline{B}^+_{C_2}})^+ \Longrightarrow \begin{cases} true(A)^+ \\ true(B)^+ \end{cases} \tag{4.2}$$

This rule rewrites a goal of the form $true(A \wedge B)$ into two subgoals $true(A)$ and $true(B)$. This is exactly the effect produced by the original inference rule if used to refine a conjunction-goal. The discussion of how to use this kind of rule is deferred until Chapter 7.

The rule for $\vee_e$ is one of the rules that can be simplified in Step 3. This rule has a place-holder expression ($true(C)$) to match and preserve the goal while some hypothesis is eliminated. This type of rule is common in Natural Deduction style presentations of logics. The simplification of the rule consists of identifying this fact and converting the rule into one where the disjunction ($true(A \vee B)$) in a hypothesis (negative polarity) is rewritten into $true(A)$ and $true(B)$ as in the rule $\wedge_i$ mentioned above. Again we obtain a twin-rule:

$$true(\boxed{\underline{A}^-_{C_1} \vee \underline{B}^-_{C_2}})^- \Longrightarrow \begin{cases} true(A)^- \\ true(B)^- \end{cases} \tag{4.3}$$

Context Rewrite Rules

Context rules (rewrite or wave) are rules where a new fresh variable is introduced in the rewritten term. The name of the variable depends on the context where the rewriting takes place. For example, the rule for existential elimination in natural deduction style predicate logic, encoded in LF as:

$$\Pi_{p:i\to o}\Pi_{q:o} true(\exists(p)) \to (\Pi_{t:i} true(p(t)) \to true(q)) \to true(q)$$

can be simplified as we did in the last section with rule $\vee_e$ to obtain the following rewrite rule:

$$true(\exists P^-)^- \Longrightarrow true(P^-(\hat{t}))^-$$

The expression $\hat{t}$ stands for a new variable fresh in the context at the time of the application of the rule. Variable $\hat{t}$ is generated when the rule is applied.

To see why this is needed we need to look at how the inference rule is applied. First, the inference rule is forward-chained with some hypothesis involving $\exists$, $\exists r$ say. This will generate a new hypothesis $h_1 : (\Pi_{t::i} true(r(t)) \rightarrow true(q)) \rightarrow true(q)$. Then this hypothesis is used to refine the goal, $true(z)$ say, and we obtain a new goal: $(\Pi_{t::i} true(r(t)) \rightarrow true(z))$. Finally, introducing Π and $\rightarrow$ we obtain the original goal $true(z)$ and a new hypothesis $true(r(\hat{t}))$ where $\hat{t}$ is a new variable of type i in the context.

Now, we will see an example theorem that uses the simplest rules mentioned above.

4.3 A Simple Example

This example is from propositional logic. We will use in the proof some of the wave rules introduced in the last section. The statement is:

$$true((a \supset b) \wedge (b \supset c) \supset (a \supset c))$$

After providing polarity values, the system balances the sequent. Constants a and c make connections across the sequent symbol. Difference unification of goal and hypothesis gives us the wave annotation that mark the two skeletons (colours c_1 and c_2) that need to be rippled to make the connections:

$$\begin{array}{lll} hyp_1 & : & true(\boxed{\underline{a}^{+}_{\{c_1\}} \supset b^{-}}^{-}_{\{c_1\}} \wedge \boxed{b^{+} \supset \underline{c}^{-}_{\{c_2\}}}^{-}_{\{c_2\}})^{-}_{\{c_1,c_2\}} \\ \vdash & & true(\boxed{\underline{a}^{-}_{\{c_1\}} \supset \underline{c}^{+}_{\{c_2\}}}_{\{c_1,c_2\}})^{+} \end{array} \tag{4.4}$$

The next step is to ripple the hypotheses. First, Rule **wr-**$\wedge_{\{el\}}$ is applied:

$$
\begin{array}{lll}
hyp_1 & : & true(\boxed{(a^+ \supset b^-)^- \wedge \boxed{b^+ \supset \underline{c}^-_{\{c_2\}}}^-_{\{c_2\}}}^{\{c_2\}}\ )^- \\
hyp_2 & : & true(\boxed{\underline{a}^+_{\{c_1\}} \supset b^-}^{\{c_1\}}\ )^- \\
\\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}}\ )^+
\end{array}
\tag{4.5}
$$

The annotations corresponding to the colour just rippled are removed from hyp_1. Now rule **wr-**$\wedge_{\{er\}}$ is applied to hyp_1 to ripple c_2:

$$
\begin{array}{lll}
hyp_1 & : & true((a^+ \supset b^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_2 & : & true(\boxed{\underline{a}^+_{\{c_1\}} \supset b^-}^{\{c_1\}}\ )^- \\
hyp_3 & : & true(\boxed{b^+ \supset \underline{c}^-_{\{c_2\}}}^{\{c_2\}}\ )^- \\
\\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}}\ )^+
\end{array}
\tag{4.6}
$$

Rippling continues using a weakened version of Rule **wr-**$\supset_e$ on hyp_2:

$$
\begin{array}{lll}
hyp_1 & : & true((a^+ \supset b^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_2 & : & true(a^+ \supset b^-)^- \\
hyp_3 & : & true(\boxed{b^+ \supset \underline{c}^-_{\{c_2\}}}^{\{c_2\}}\ )^- \\
hyp_4 & : & \boxed{\underline{true(a)}^+_{\{c_1\}} \rightarrow true(b)^-}^{\{c_1\}} \\
\\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}}\ )^+
\end{array}
\tag{4.7}
$$

and then on hyp_3:

$$
\begin{array}{lll}
hyp_1 & : & true((a^+ \supset b^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_2 & : & true(a^+ \supset b^-)^- \\
hyp_3 & : & true(b^+ \supset c^-)^- \\
hyp_4 & : & \boxed{\underline{true(a)}^+_{\{c_1\}} \rightarrow true(b)^-}^{\{c_1\}} \\
hyp_5 & : & \boxed{true(b)^+ \rightarrow \underline{true(c)}^-_{\{c_2\}}}^{\{c_2\}} \\
\\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}}\ )^+
\end{array}
\tag{4.8}
$$

At this point there is no more rippling possible in the hypotheses so rippling starts in the goal. The rule applied is **wr-**$\supset_i$:

$$\begin{array}{lll} hyp_4 & : & \boxed{\underline{true(a)}^{+}_{\{c_1\}} \rightarrow true(b)^{-}}_{\{c_1\}} \\ hyp_5 & : & \boxed{true(b)^{+} \rightarrow \underline{true(c)}^{-}_{\{c_2\}}}_{\{c_2\}} \\ & \vdash & \boxed{\underline{true(a)}^{-}_{\{c_1\}} \rightarrow \underline{true(c)}^{+}_{\{c_2\}}}^{+}_{\{c_1,c_2\}} \end{array} \tag{4.9}$$

Now both colours are fully rippled and fertilisation is possible with hyp_5 leaving the following goal:

$$\begin{array}{lll} hyp_4 & : & true(a)^{+} \rightarrow true(b)^{-} \\ hyp_5 & : & true(b)^{+} \rightarrow true(c)^{-} \\ hyp_6 & : & true(a)^{-} \\ & \vdash & true(b)^{+} \end{array} \tag{4.10}$$

Fertilisation is again possible using hypothesis hyp_4:

$$\begin{array}{lll} hyp_4 & : & true(a)^{+} \rightarrow true(b)^{-} \\ hyp_5 & : & true(b)^{+} \rightarrow true(c) \\ hyp_6 & : & true(a)^{-} \\ & \vdash & true(a)^{+} \end{array} \tag{4.11}$$

This sequent is an axiom and the proof is finished.

Using all possible rule applications generates a big search space. At each stage of the proof above there are other rules we could have applied. The restriction that in Rippling we only use rules which can make sense as wave rules and that to be applied their annotations need to match those of the expression, reduces the space of applicable rules to only a few. For instance, in the example above, rules **rw-**$\wedge_{el}$ and **rw-**$\wedge_{er}$ are applicable to hyp_1, **rw-**$\supset_e$ is applicable to hyp_2 and hyp_3 and **rw-**$\supset_i$ to the goal. The use of wave rules instead of rewrites permits to reduce the number of rules applicable to a given sequent. In sequent 4.8 for instance, the only wave rule applicable is **wr-**$\supset_i$ although the rest of the rewrites just mentioned is applicable. The number of rule applications possible in this sequent is reduced from 5 to 1 by the use of wave rules. We will analyse search issues in Chapter 9.

4.4 Theoretical Issues

We have given in this chapter an overview of the techniques we propose in this thesis. We now address some theoretical issues concerning these techniques.

4.4.1 Soundness

The question of whether our system produces sound proofs has two aspects to it. The first one is the absolute one: is every proof of the system correct?

The techniques we propose are embedded in a proof planning environment. As such, there is a planning program coupled with a proof editor where the plans created by the planner are executed. It is only at this final stage that the actual proofs are constructed by tactics. The tactics contribute to the actual proof only by applying inference rules from the framework logic so our system is sound with respect to LF.

Our system is sound in an absolute sense: every proof produced by the system is correct with respect to the object logic, provided that the encoding of that logic is *faithful*. A faithful encoding of an object-logic in a framework logic is such that for every proof in the framework logic using the encoding there is a proof in the logic for the corresponding theorem. See [Harper *et al* 92] for a description of this with respect to LF.

The second aspect of soundness we mentioned is more interesting and has to do with the relationship between planner and proof editor. Since we intend to have all the proof search process at the planning level, we expect the proof plan produced to be applicable in the proof editor. We don't expect the interaction between planner and proof editor to behave in a trial and error fashion: "create a plan and see if it is applicable; if not create another one and try again". So, the question is: Are the proof plans produced by the planner always executable in the proof editor?

The answer is no, but where can the problems be? In Section 4.2 we sketched the four main steps of the system. In Balance, Π-introduction and object level introduction rules are applied. This usually presents no problems. The specification of these process in the methods is accurate enough and the part of the plan involving balance is always applicable in the proof editor in our experience.

Comparison does not correspond to any direct action in the proof editor. In rippling, wave rules correspond to tactics that cut into the proof the rewritten expression and then "justify" the cut through various strategies. This justification of the cut de-

pends on the applicability of the inference rule corresponding to the wave rule to the expression being rewritten. At the planning level this is guaranteed by the polarity annotations. The properties of polarity at the framework level proven in Chapter 5 ensure this.

Improper rules introduce meta-variables into the planning process. These variables are instantiated at the weak-fertilisation stage by unification. There could be two problems with this. Meta-variables may be instantiated to other meta-variables in the plan without telling the proof editor what the appropriate object to be instantiated is (See example 7 in Chapter 9). The other problem with meta-variables is that they may be instantiated to an object which is not well-formed in the logic. The system does not keep typing information at the planning level and does not verify the well-formedness of the objects. This is usually done in the proof editor (see Examples 8 and 9 in Chapter 9). In neither of the two cases the plan would be applicable.In Section 9.4.6 we discuss these issues.

This problem is not exclusive to our system. In systems where meta-variables may be introduced in the proof (e.g. LEGO, Section 2.2), at the end of a proof, when all the meta-variables are instantiated, the context of some sequents may not be well-formed.

Weak-fertilisation corresponds to forward and backward chaining of object level inference rules. This process presents no problems for the proof editor.

The last step, unblock, is designed to apply rewrite and inference rules to try to unblock the difference reduction mechanism. Inference rules may introduce meta variables; improper rewrite rules also introduce meta variables. As with rippling, if these variables are not properly instantiated, the plan won't be applicable.

To summarise, the main problems arising between planner and proof editor come from the introduction of meta-variables. In any case, soundness always depends on having an faithful encoding of the logic so, in the rest of the thesis, we will assume that the encoding of the logic we are using is faithful.

4.4.2 Termination

The problems for the termination of our system also come from improper wave rules. Balance can only introduce a finite number of times, so it always terminates. Weak-fertilise also has a finite number of hypotheses to operate and as soon as the hypotheses used are always different, weak-fertilise will terminate.

In rippling, both proper and improper wave rules are terminating but, when unblocking, rewrite rules may not be measure decreasing, inference rules are applied arbitrarily and so there is no guarantee of termination.

4.4.3 Completeness

Rewriting on its own is not complete. If we consider, for instance, the implicative part of minimal logic, we have as rewrites rules **rw-**$\supset_i$ and **rw-**$\supset_e$. If we rewrite the following theorem:

$$\vdash true(a \supset b \supset a)$$

we obtain:

$$\vdash true(a) \rightarrow true(b) \rightarrow true(a)$$

In this sequent there are no more applicable rewrites and there is no way of introducing formulae from the goal into the hypotheses side to obtain an axiom. Even having introduction and rewriting is not enough. The theorem:

$$\vdash true(a \supset (a \supset b) \supset b)$$

can be rewritten exhaustively to obtain:

$$\vdash true(a) \rightarrow (true(a) \rightarrow true(b)) \rightarrow true(b)$$

Introducing as much as possible gives:

$$\begin{aligned} hyp_1 &: true(a) \\ hyp_2 &: true(a) \rightarrow true(b) \\ &\vdash true(b) \end{aligned}$$

and this is still not an axiom. Refinement is required. Weak-fertilisation and balance account for refinement and introduction. Rippling needs introduction to have some hypotheses to compare the goal with and have some annotation. However, as in the previous example, it also needs refinement.

The question of completeness also hinges on the encoding of the logic being *adequate*. That is, for every proof in the logic there is a proof in the framework logic using the encoding [Harper *et al* 92].

Our methodology starts by using strong heuristics that involve little search but then constraints on rule application are relaxed as required until search becomes completely unconstrained. Rippling is the most constrained of the heuristics. As we saw, it does rewriting in a well-directed manner. Rippling is not powerful enough to contend with arbitrary situations, however. As we will see in Chapter 8, we use a method called *unblock* that first relaxes restrictions on rewrites to try to drive the proof planning process back to a situation where rippling can be used again, and then, failing that, it allows free application of inference rules. This process can be seen as one of strong heuristic guidance with the option of "graceful" degradation into unconstrained search.

Unconstrained search would give us completeness in the sense that, with the appropriate search strategy, a proof plan for any provable statement could be produced. Unconstrained search here refers to the ability of the system to apply arbitrary rules from a signature however. This improves the power of the system but, since the inference rules of a signature are applied via tactics and there is no access to direct LF inference rule application, we still have no guarantee that the system is complete.

4.5 Summary

In this chapter we have outlined the techniques we propose in this thesis. We expanded on the idea of rippling, used in inductive theorem proving, to build a methodology to automate proof search in several logics. We use Natural Deduction presentations because from them it is possible to extract wave rules.

Brief explanations of the main concepts involved were given, namely: polarity, polarised difference unification, fertilisation and wave rules extraction. We presented a step-by-

step description of a simple example from Intuitionistic Propositional Logic to illustrate and motivate our techniques. We hope this chapter will provide the reader with an insight into the problems addressed and the solutions proposed. The next chapters describe more technically each one of the issues discussed here.

Chapter 5

Polarity

In the previous chapter, in Section 4.2.1, we introduced a concept of polarity. Now, we will cover in more detail important topics related to polarity. We will make reference to LF in all our definitions and examples because this agrees with the system design described in Chapter 8, but the concepts can be easily extended or adapted to other framework logics.

5.1 Polarity in LF

We denote a polarity assignment of polarity p to term T by T^p. p could take values $+$ or $-$. We now define the polarity of the subjudgements.

Definition 1 *If J is an LF judgement, and if J as been assigned polarity p (i.e. J^p), we define the polarity assignments of its sub-judgements recursively on its form as follows:*

- *If $(J_1 \rightarrow J_2)^p$ then ${J_1}^{\bar{p}}$ and ${J_2}^p$.*
- *If $(\Pi_{x:J_1} J_2(x))^p$ then ${J_2}^p$*

where $\bar{p}$ denotes the polarity opposite to the polarity denoted by p. Since the polarity values of subjudgements depend on the polarity assigned to the terms they are contained in, we use the notation $J_1[{J_2}^q]^p$ to denote the occurrence of J_2 with polarity p as a subterm of J_1 when it has polarity p.

Property 1 *Let J_1 and J_2 be LF judgements and p and q polarity values. If $J_1[J_2{}^q]^p$ then $J_1[J_2{}^{\bar{q}}]^{\bar{p}}$.*

Proof: Immediate consequence of Definition 1.□

In LF, as a convention, we assign a positive polarity to the goal and a negative polarity to each hypothesis. The main property we obtain from polarity is the following. We denote by $\vdash_{lf}$ derivability in LF system L (Figure A.3) when the signature Σ is empty.

Property 2 *Let J_1 and J_2 be LF judgements. For any LF judgements F and G,*

1. *If $\Gamma \vdash_{lf} J_1 \to J_2$ then $\Gamma, h : F[J_1^-]^- \vdash_{lf} F[J_2^+]^+$*
2. *If $\Gamma \vdash_{lf} J_1 \to J_2$ then $\Gamma, h : G[J_2^+]^- \vdash_{lf} G[J_1^-]^+$*

Proof: Let's assume that $\Gamma \vdash_{lf} J_1 \to J_2$. We do induction on the structure of F and G. First, if F and G are atomic judgements, then since in Property 2.2 J_2 cannot occur positively in the hypotheses, we need only verify Property 2.1:

$$\Gamma, h : J_1 \vdash_{lf} J_2$$

which is immediate from our assumption.

Now let's assume that for some F:

$$\text{If } \Gamma \vdash_{lf} J_1 \to J_2 \text{ then } \Gamma, h : F[J_1^-]^- \vdash_{lf} F[J_2^+]^+ \tag{5.1}$$

$$\text{If } \Gamma \vdash_{lf} J_1 \to J_2 \text{ then } \Gamma, h : G[J_2^+]^- \vdash_{lf} G[J_1^-]^+ \tag{5.2}$$

We now prove that:

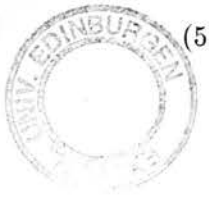
$$\Gamma, h : (K \to F[J_1^-]^-)^- \vdash_{lf} (K \to F[J_2^+]^+)^+ \tag{5.3}$$

$$\Gamma, h : (F[J_1^-]^+ \to K)^- \vdash_{lf} (F[J_2^+]^- \to K)^+ \tag{5.4}$$

$$\Gamma, h : (\Pi_{x:J_1} F[J_1(x)^-]^-)^- \vdash_{lf} (\Pi_{x:J_1} F[J_2(x)^+]^+)^+ \tag{5.5}$$

$$\Gamma, h : (K \to G[J_2^+]^-)^- \vdash_{lf} (K \to G[J_1^-]^+)^+ \tag{5.6}$$

$$\Gamma, h : (G[J_2^+]^+ \to K)^- \vdash_{lf} (G[J_1^-]^- \to K)^+ \tag{5.7}$$



$$\Gamma, h : (\Pi_{x:J_1} G[J_2(x)^+]^-)^- \vdash_{lf} (\Pi_{x:J_1} G[J_1(x)^-]^+)^+ \tag{5.8}$$

The first three formulae above correspond to the step case for Property 2.1 and the last three correspond to the step case for Property 2.2.

To prove Formula 5.3, we can build the following backwards derivation:

$$\begin{array}{rl}
\Gamma, h : (K \to F[J_1^-]^-)^- \vdash_{lf} (K \to F[J_2^+]^+)^+ & \text{(goal)} \\
\Gamma, h : (K \to F[J_1^-]^-)^-, k : K \vdash_{lf} F[J_2^+]^+ & (\to r) \\
\Gamma, h : (K \to F[J_1^-]^-)^-, k : K \vdash_{lf} F[J_1^+]^+ & (\text{cut}(F[J_1^+]^+) \text{ and 5.1}) \\
\Gamma, h : (K \to F[J_1^-]^-)^-, k : K \vdash_{lf} K & \text{(refine hyp)} \\
\Box & \text{(axiom)}
\end{array}$$

For 5.4

$$\begin{array}{rl}
\Gamma, h : (F[J_1^-]^+ \to K)^- \vdash_{lf} (F[J_2^+]^- \to K)^+ & \text{(goal)} \\
\Gamma, h_1 : (F[J_1^-]^+ \to K)^-, h_2 : F[J_2^+]^- \vdash_{lf} K & (\to) \\
\Gamma, h_1 : (F[J_1^-]^+ \to K)^-, h_2 : F[J_2^+]^- \vdash_{lf} F[J_1^-]^+ & \text{(refine hyp)} \\
\Box & (5.2)
\end{array}$$

For 5.5:

$$\begin{array}{rl}
\Gamma, h : (\Pi_{x:J_1} F[J_1(x)^-]^-)^- \vdash_{lf} (\Pi_{x:J_1} F[J_2(x)^+]^+)^+ & \text{(goal)} \\
\Gamma, h : (\Pi_{x:J_1} F[J_1(x)^-]^-)^-, a : J_1 \vdash_{lf} F[J_2(a)^+]^+ & (\Pi r) \\
\Gamma, h : (\Pi_{x:J_1} F[J_1(x)^-]^-)^-, a : J_1 \vdash_{lf} F[J_1(a)^+]^+ & (\text{cut}(\ F[J_1(a)^+]^+) \text{ and 5.1}) \\
\Gamma, h_1 : (\Pi_{x:J_1} F[J_1(x)^-]^-)^-, a : J_1, h_2 : F[J_1(a)^-]^- \vdash_{lf} F[J_1(a)^+]^+ & (\Pi l) \\
\Box & \text{(axiom)}
\end{array}$$

For 5.6:

$$\begin{array}{rl}
\Gamma, h : (K \to G[J_2^+]^-)^- \vdash_{lf} (K \to G[J_1^-]^+)^+ & \text{(goal)} \\
\Gamma, h : (K \to G[J_2^+]^-)^-, k : K \vdash_{lf} G[J_1^-]^+ & (\to r) \\
\Gamma, h : (K \to G[J_2^+]^-)^-, k : K \vdash_{lf} G[J_2^-]^+ & (5.2) \\
\Gamma, h : (K \to G[J_2^+]^-)^-, k : K \vdash_{lf} K & \text{(refine hyp)} \\
\Box & \text{(axiom)}
\end{array}$$

For 5.7

$$\begin{array}{rl}
\Gamma, h : (G[J_2^+]^+ \to K)^- \vdash_{lf} (G[J_1^-]^- \to K)^+ & \text{(goal)} \\
\Gamma, h_1 : (G[J_2^+]^+ \to K)^-, h_2 : G[J_1^-]^- \vdash_{lf} K & (\to r) \\
\Gamma, h_1 : (G[J_2^+]^+ \to K)^-, h_2 : G[J_1^-]^- \vdash_{lf} G[J_2^+]^+ & \text{(refine hyp)} \\
\Box & (5.1)
\end{array}$$

For 5.8

$$\begin{array}{rl}
\Gamma, h : (\Pi_{x:J_1} G[J_2(x)^+]^-)^- \vdash_{lf} (\Pi_{x:J_1} G[J_1(x)^-]^+)^+ & \text{(goal)} \\
\Gamma, h : (\Pi_{x:J_1} G[J_2(x)^+]^-)^-, a : J_1 \vdash_{lf} G[J_1(a)^-]^+ & (\Pi r) \\
\Gamma, h_1 : (\Pi_{x:J_1} G[J_2(x)^+]^-)^-, a : J_1, h_2 : G[J_2(a)^+]^-)^- \vdash_{lf} G[J_1(a)^-]^+ & (\Pi l) \\
\Box & (5.2)
\end{array}$$

□

This property of polarity guarantees a kind of *structural monotonicity* that allows us to justify the use of inference rules as rewrite rules. This is captured in the next property.

Property 3 *If* $\Gamma \vdash_{lf} J_1 \rightarrow J_2$, *then for all judgements* H,G,

1. *If* $\Gamma \vdash_{lf} G[J_1^+]^+$ *then* $\Gamma \vdash_{lf} G[J_2^+]^+$
2. *If* $\Gamma \vdash_{lf} G[J_2^-]^+$ *then* $\Gamma \vdash_{lf} G[J_1^-]^+$
3. *If* $\Gamma, h : H[J_1^+]^- \vdash_{lf} G$ *then* $\Gamma, h : H[J_2^+]^- \vdash_{lf} G$
4. *If* $\Gamma, h : H[J_2^-]^- \vdash_{lf} G$ *then* $\Gamma, h : H[J_1^-]^- \vdash_{lf} G$

Proof: All formulae are direct consequence of Property 2. □

We can generalise the property to iterated inference rules to rewrite positive occurrences of terms (Property 4 below). We first introduce some lemmas in order to prove the property. In order to prove the lemmas we will make use of some of LF's admissible rules *Transitivity* and *Weakening* [Harper *et al* 92] ($\vdash_\Sigma$ denotes derivability in LF's system L (as in Figure A.3) with respect to signature Σ):

$$\text{If } \Gamma \vdash_\Sigma M : A \text{ and } \Gamma, x : A, \Gamma' \vdash_\Sigma \alpha \text{ then } \Gamma, [M/x]\Gamma' \vdash_\Sigma [M/x]\alpha \tag{5.9}$$

$$\text{If } \Gamma \vdash_\Sigma \alpha \text{ then } \Gamma, \Gamma' \vdash_\Sigma \alpha \tag{5.10}$$

Lemma 1 *If* A *and* B *are LF types then the following are equivalent:*

(*i*) $\Gamma, a : A \vdash_{lf} B(a)$

(*ii*) $\Gamma \vdash_{lf} \Pi_{x:A} B(x)$

Proof:

(*i*) $\Rightarrow$ (*ii*) This is LF's Πr rule.

(*ii*) $\Rightarrow$ (*i*) Let us assume that (*ii*) holds. We use LF's admissible rule 5.9 backwards on i to cut $\Pi_{x:A} B(x)$ in. We now need to prove two sequents:

$$\Gamma, a : A \vdash_{lf} \Pi_{x:A} B(x)$$

$$\Gamma, a : A, h : \Pi_{x:A} B(x) \vdash_{lf} B(a)$$

For the first one we apply LF's admissible rule 5.10 and obtain our assumption *ii*. For the second, we apply LF's rule Πl backwards to obtain:

$$\Gamma, a : A, h_1 : \Pi_{x:A} B(x), h_2 : B(a) \vdash_{lf} B(a)$$

which is an axiom.□

Lemma 2 *Let J_1, J_2 and L be LF judgements and Γ an LF context. If $\Gamma \vdash_{lf} F[J_1^+]^+$ and $\Gamma \vdash_{lf} F[(J_1 \rightarrow J_2)^+]^+$ then $\Gamma \vdash_{lf} F[J_2^+]^+$*

Proof: We do induction on the structure of F. First, we assume F is atomic, then we have to prove:

$$\Gamma \vdash_{lf} J_1^+ \text{ and } \Gamma \vdash_{lf} (J_1 \rightarrow J_2)^+$$

then

$$\Gamma \vdash_{lf} J_2^+$$

but this is an instance of LF's $\rightarrow r$ rule so we proceed to the step case. We first assume that the theorem holds for F and prove it for the cases where F is embedded in a more complex expression involving Π. We now have to prove:

$$\Gamma \vdash_{lf} (\Pi_{x:K} F[J_1^+]^+)^+ \text{ and } \Gamma \vdash_{lf} (\Pi_{x:K} F[(J_1 \rightarrow J_2)^+]^+)^+$$

then

$$\Gamma \vdash_{lf} (\Pi_{x:K} F[J_2^+]^+)^+$$

Applying Lemma 1 to the two antecedents above we obtain:

$$\Gamma, k : K \vdash_{lf} F[J_1^+]^+$$

$$\Gamma, k : K \vdash_{lf} F[(J_1 \rightarrow J_2)^+]^+$$

Using the induction hypothesis we can conclude:

$$\Gamma, x : K \vdash_{lf} F[J_2^+]^+$$

which is equivalent to the goal we want to obtain, again, by Lemma 1. □

We now look at the generalised property we mentioned before.

Property 4 *Let $J, J_1, \ldots, J_n$ be judgements and suppose:*

$$\Gamma \vdash_{lf} J_1 \to (J_2 \to \cdots \to (J_n \to J))$$

Then:

$$\textit{If} \left(\begin{array}{c} \Gamma \vdash_{lf} F[J_1^+]^+ \\ \Gamma \vdash_{lf} F[J_2^+]^+ \\ \vdots \\ \Gamma \vdash_{lf} F[J_n^+]^+ \end{array} \right) \textit{ then } \Gamma \vdash_{lf} F[J^+]^+$$

Proof: We prove the property by induction on n. Let's assume that $\Gamma \vdash_{lf} J_1 \to (J_2 \to \ldots \to (J_n \to J))$. When $n = 1$, the theorem is reduced to Property 3(1). Let's now assume that the theorem holds for an arbitrary n. We now prove that if $\Gamma \vdash_{lf} J_1 \to (J_2 \to \ldots \to (J_n \to (J_{n+1} \to J)))$ and

$$\overrightarrow{\Gamma \vdash_{lf} F[J_{n+1}^+]^+}$$

then

$$\Gamma \vdash_{lf} F[J^+]^+ \tag{5.11}$$

Using the induction hypothesis we can conclude that:

$$\Gamma \vdash_{lf} F[(J_{n+1} \to J)^+]^+$$

and from Lemma 2 we have the conclusion 5.11. □

The following property is a corollary of the one we've just proven. It states the same rewriting property for the hypotheses.

Property 5 *With the same assumptions from the previous property, the following holds:*

$$\textit{If} \left(\begin{array}{c} \Gamma, h : F[J_1^+]^- \vdash_{lf} L \\ \Gamma, h : F[J_2^+]^- \vdash_{lf} L \\ \vdots \\ \Gamma, h : F[J_n^+]^- \vdash_{lf} L \end{array} \right) \textit{ then } \Gamma, h : F[J^+]^- \vdash_{lf} L$$

Proof: From Lemma 1 the previous sequent is equivalent to:

$$\text{If} \left(\begin{array}{c} \Gamma \vdash_{lf} F[J_1^+]^- \to L^+ \\ \Gamma \vdash_{lf} F[J_2^+]^- \to L^+ \\ \vdots \\ \Gamma \vdash_{lf} F[J_n^+]^- \to L^+ \end{array} \right) \text{ then } \Gamma \vdash_{lf} F[J^+]^- \to L^+$$

and this follows from Property 4.□

These properties justify the use of *twinrules* introduced in Chapter 7 below.

5.2 Object Level Polarity

In Chapter 4 we showed how difference unification is used to find potential connections between hypotheses and conclusion. We also saw how polarity annotations at the object level are used to restrict difference unification to consider only skeletons whose polarities are compatible.

Using difference unification to compare hypotheses and goal is not enough to spot connections. Expressions may not be able to make connections after they have been rippled. For example, the annotations in the following sequent:

$$\begin{array}{rcl} hyp_1 & : & true(\boxed{\underline{a}_{\{c_1\}} \supset b}) \\ & \vdash & true(a) \end{array}$$

indicate that the skeleton $true(a)$ in the hypothesis can be rippled to make a connection with the goal. Rippling hyp_1 with (a weakened version of) rule **wr-**$\supset_e$ (Section 4.2) gives:

$$\begin{array}{rcl} hyp_1 & : & true(a \supset b) \\ hyp_2 & : & \boxed{\underline{true(a)}_{\{c_1\}} \rightarrow true(b)} \\ & \vdash & true(a) \end{array}$$

Now hyp_2 is fully rippled but $true(a)$ in hyp_2 has the same polarity to the goal:

$$\begin{array}{rcl} hyp_1 & : & true(a \supset b)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ & \vdash & true(a)^+ \end{array}$$

and so cannot be used to simplify the sequent (make a connection). At this point we must backtrack to the point of difference unification and try a different unification or a different hypothesis.

To avoid this problem, we use object level polarity annotations. They represent a partial evaluation procedure to preview the polarity of the judgement dominating an object level expression after rewriting. For example, an expression of the form

$$true(a \supset b)^-$$

can only be rewritten with the rewrite rules in Section 4.2 as:

$$true(a)^+ \rightarrow true(b)^-$$

We represent this fact by adding annotations to a and b as follows:

$$true(a^+ \supset b^-)^-$$

We assume that the polarity of the top object expression $(a \supset b)$ is the same as that of the whole judgement $(true(a \supset b))$ and we don't write it down. Only subexpressions of the top object expressions are annotated with polarity.

This polarity annotation indicates to the difference unifier beforehand that a in the hypothesis cannot make a connection with a in the goal because they both have the same polarity. This restriction saves a considerable amount of search.

Some object level expressions can be rewritten in different ways leading to different polarity assignments. For example, the expression:

$$true(a \equiv b)^+$$

can be rewritten by rule:

$$true(a \equiv b)^+ \Longrightarrow \begin{cases} true(a \supset b)^+ \\ true(b \supset a)^+ \end{cases} \qquad (rw - \equiv_i)$$

and we will have two a and two b. We can see from the example of the implication above that, in the antecedent, a is assigned a positive polarity and b a negative polarity;

in the consequent, the polarities are the opposite for the two constants. Since in this case a and b reach different polarity values in two different branches, we assign the value $\pm$ in 5.2 to both constants, as follows:

$$true(a^{\pm} \equiv b^{\pm})$$

The algorithm to obtain the polarity of subexpression e of o in polarised judgement $J(o)^p$ is the following.

1. Let S be the set $\{J(o)^p\}$.
2. Let S_1 be the set of all possible one-step rewritings of the members of S using the set of rewrite rules extracted from a logic presentation.
3. Let S_2 be the set of atomic judgements in S_1 containing e as subexpression.
4. If not all members of S_2 are of the form $J(e)^q$ for some polarity q, let S be now S_2 and go to Step 2.
5. If S_2 is a singleton $\{J(e)^q\}$ then the polarity value of e in $J(o)^p$ is q. If S_2 is not a singleton, then $\pm$ is the polarity value of e in $J(o)^p$.

This algorithm terminates for the usual Natural Deduction presentations, in particular the ones we use in this thesis. However, we have no guarantee that it is terminating in general.

The use of polarity annotations at the object level does not guarantee that two matching expressions on both sides of the sequent with opposite polarity will make a connection. What it does tell us is that two expressions, one on each side of the sequent, which have the same polarity, will never make a connection even if they match.

This will become more clear if we look at the following:

1. If we ignore the object level and look at LF types, connecting judgements always have opposite polarity.
2. According to our definition of object level polarity, each rewrite rule application will preserve object level polarity.

3. Every wave rule application reduces the size of object level expressions in the judgements.

4. When there are two connecting judgements, their object expressions must have the same polarity they had earlier in the proof, so they must have had different object level polarity from the start.

The heuristic of using polarity annotations is useful because we use Natural Deduction presentations of logics where proofs are usually built by making connections. For this reason, we can assume that using polarity annotations in this type of presentation to identify connections is a heuristic that prunes the search-tree without compromising provability.

5.3 Summary

In this chapter we have defined polarity for LF judgements and given some of its properties. We also presented an algorithm to annotate with polarity expressions at the object level. The properties we described are important to justify the use of twin rules introduced in Chapter 4 and explained in more detail in Chapter 7 below. They also motivate the use of object level polarity to restrict difference unification to skeletons with compatible polarities.

Chapter 6

Polarised Coloured Difference Unification

In this chapter we define formally the concepts of polarised annotated term and the algorithm for polarised coloured difference unification (PCDU) mentioned in Chapter 4. The first section defines the set of annotated terms with respect to a set of colours and explains the differences with versions given elsewhere. Section 6.2 presents the specification of *polarised coloured difference unifiability.* Section 6.3 states the PCDU algorithm and Section 6.4 describes some of the properties of PCDU.

6.1 Coloured-Annotated Terms

A polarised term is a term that has been assigned a polarity value; this is represented by a term of the form: t^-, t^+, $t^\pm$ where t is a regular term. We call a term algebra with polarity values assigned to all terms and subterms, a *free polarised term algebra.* We will use the function $pol(p)$ to refer to the polarity of the polarised term p and function $term(p)$ to refer to the term resulting from removing the polarity value from polarised term p.

The following is a definition of the set of polarised coloured annotated terms (*pcats*). We define the syntax of polarised terms with wave annotations:

Definition 2 *Let Σ be a signature; let $P = PTerm(\Sigma)$ be the free polarised term algebra over Σ, VP a set of variables and p a polarity sign; COL a set of colours.*

We inductively define a hierarchy of sets of coloured-annotated terms, AT_C, indexed by colour sets $C \subseteq COL$.

- *if* $X^p \in VP$ *then* $\underline{X^p}_C \in AT_C$.
- *if* $t^p \in P$ *then* $\underline{t^p}_C \in AT_C$.
- *if* $at_i \in AT_C$ *then* $\underline{f(at_1, \ldots, at_n)^p}_C \in AT_C$
- *if* $at_i \in AT_{C_i}$, $\bigcup_i^n C_i = C$, *then* $\underline{\boxed{f(at_1, \ldots, at_n)^p}}_C \in AT_C$
- *Nothing else is in* AT_C.

There are several differences between the treatment given to annotated terms in [Bundy *et al* 90b], [Bundy *et al* 93], [Yoshida *et al* 94], [Basin & Walsh 93] and the one we use here.

Annotated terms have been formalised by giving a list of 'term addresses' that point to all the wave holes in a term. The notation of underlined terms in boxes is merely a visual device to understand better what the formal objects mean. In our approach we have decided to define annotated terms using the underlined-term-in-box notation because we find it clearer and it allows us to add more structure to terms (i.e. polarities and colours).

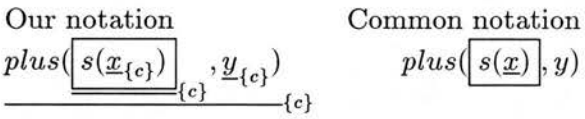
We have also changed the shape of annotated terms in the common notation to make it more uniform and we regard many forms of the common notation as abbreviations of the standard form introduced here. In other words, our standard definition of annotated terms is compatible with the common one used elsewhere. In the following table, we list the differences between our notation and the common notation and we consider the latter an abbreviation of the former.

Our notation	Common notation	Proviso
$\underline{t}_C$	t	[a]
$\underline{f(\underline{t_1}_C, \ldots, \underline{t_n}_C)}_C$	$\underline{f(t_1, \ldots, t_n)}_C$	
$\underline{t}_\emptyset$	t	
$\underline{t}_{\{c\}}$	$\underline{t}$	[b]

[a] If $\underline{t}_C$ does not occur inside a wave front.

[b] If c is the only colour occurring in the expression containing the term.

An example of the abbreviation is the following:



The second abbreviation allows us to abbreviate the holes of the arguments of *plus*. The first one allows us to remove the top-most hole and the last equivalence ignores the colour since there is only one in the whole expression.

Definition 3 *Given an annotated term as above, we can define what its skeleton is. The skeletons of a term are parameterised by colours. If c is a colour, we define:*

- $skel(\underline{t^p}_C, c) = \{\}$ *if* $c \notin C$
- $skel(\underline{X^p}_C, c) = \{X^p\}$ *if* $c \in C, X^p \in VP$.
- $skel(\underline{t^p}_C, c) = \{t^p\}$ *if* $c \in C, t^p \in P$.
- $skel(\underline{f(at_1, \ldots, at_n)^p}_C, c) = \{f(s_1, \ldots, s_n)^p | s_i \in skel(at_i, c)\}$ *if* $c \in C$.
- $skel(\underline{\boxed{f(at_1, \ldots, at_n)^p}}_C, c) = skel(at_1, c) \cup \ldots \cup skel(at_n, c)$ *if* $c \in C$.

Definition 4 *The function erase removes all annotation from an annotated term:*

- $erase(\underline{X^p}_C) = X^p$
- $erase(\underline{t^p}_C) = t^p$
- $erase(\underline{f(\vec{t_n})^p}_C) = f(\overrightarrow{erase(t_n)})^p$
- $erase(\underline{\boxed{f(\vec{t_n})^p}}_C) = f(\overrightarrow{erase(t_n)})^p$

The set $AT_C/t = \{s \in AT_C \mid erase(s) = t\}$ is the set of all annotated terms with the given erasure.

6.2 Polarised Coloured Difference Unifiability

As we saw in Chapter 4, polarised annotated terms are used to express differences in structure and proof-role of two terms. The wave annotations highlight the structural variance between them while the polarity values indicate the proof context in which they occur.

For our work, we need to identify terms which constitute potential connections. We therefore need to define our difference unification algorithm to unify terms which are structurally similar as in standard difference unification, but also whose skeletons have compatible polarities. We express this more formally in the definitions below.

The symbols $\circ$ and $\bullet$ represent compatible polarities: $\{+,-\}, \{+,\pm\}, \{-,\pm\}$ and $\{\pm,\pm\}$.

Definition 5 *The relation $t_1 \doteq t_2$ over polarised terms is true if t_1 and t_2 are equal modulo polarity compatibility. That is:*

1. $t^\circ \doteq t^\bullet$
2. $f(\overrightarrow{a_n})^\circ \doteq f(\overrightarrow{b_n})^\bullet \quad \forall i. a_i \doteq b_i.$

This definition is now extended to sets of polarised terms as follows:

Definition 6 *Two sets of polarised terms P and Q are compatible modulo polarity, expressed as $P \rightleftharpoons Q$, if:*

$$P \rightleftharpoons Q \text{ if } (\forall p \in P. \exists q \in Q. p \doteq q) \wedge (\forall q \in Q. \exists p \in P. p \doteq q)$$

The following definition states when two polarised terms are pcd-unifiable. We use this definition to prove the correctness of our algorithm in Section 6.4.

Definition 7 *Two polarised terms t_1 and t_2 are pcd-unifiable if there are two annotated terms $at_1 \in AT_C/t_1$ and $at_2 \in AT_C/t_2$ for some set of colours C and a substitution τ such that for all $c \in C$*

$$skel(at_1, c)\tau \rightleftharpoons skel(at_2, c)\tau$$

There may be more than one way in which terms may be pcd-unifiable. Just as in difference unification, there may be several pair of annotated terms which fulfill the requirements of Definition 7.

The algorithm presented in the next section computes, for any two terms t_1 and t_2, all variable substitutions that fulfill Definition 7. It has been adapted from the description of the algorithm for difference unification presented in [Basin & Walsh 93].

6.3 PCDU Algorithm

The following definition gives the rules for polarised coloured difference unification. The algorithm is defined as a non-deterministic set of transformation rules applicable to *triples* $\langle \sigma, S, \tau \rangle$. σ is a substitution of annotated terms for variables; we call it an *annotated substitution.* S is a sequence of tuples $\langle a, b, A, B \rangle$, called *pcdu-problems.* a,b are terms and A,B are variables where annotated terms will be incrementally instantiated (i.e. partial annotated terms with variables will be instantiated in them as new tuples are generated). We call sequence S the *problem sequence* of the triple. τ is a variable substitution of plain terms. We call τ the *term substitution* of the triple.

Given a colour set C, the algorithm starts with $\langle \{\}, \{\langle a, b, A, B \rangle\}, \{\} \rangle$ and ends with $\langle \sigma, \{\}, \tau \rangle$. $A\sigma$ and $B\sigma$ will be the annotated terms corresponding to a and b and τ will be the term substitution of the pcd-unification.

The algorithm always gives an answer. If two terms t_1 and t_2 don't difference-unify, the resulting annotated terms are: $\underline{t_1}_{\emptyset}$ and $\underline{t_2}_{\emptyset}$. This algorithm finds all common skeletons to the two terms and assigns them a colour; therefore, if no colour is assigned the terms are not difference unifiable.

The following definition gives a set of transformation rules. They take a triple —as defined above— and produce another one. The rules have the form: $T_1 \Rightarrow T_2$ constraints: $CONS$, and denote the transformation of a triple matching T_1 into triple T_2 provided that $CONS$ hold. Constraints often rely on variables being instantiated in a state ahead of the present one, therefore, they have to be verified *post-hoc* when the information is available. They are only well-formedness constraints.

When the rules are applied exhaustively to a triple $\langle\{\}, \{\langle a, b, A, B\rangle\}, \{\}\rangle$ the final triple will contain the variable substitutions necessary to make a and b pcd-unifiable according to Definition 7. These rules are non-deterministic; the final triples given by all possible sequences of applications correspond to all possible pcd-unifications of a and b. The properties of the algorithm are explained in detail in Section 6.4.

Definition 8 *If $at_1, at_2 \in AT_C/t$ for some t, then the function $superpose(at_1, at_2)$ is defined by pattern-matching:*

1. $superpose(\underline{t^p}_{C_1}, \underline{t^p}_{C_2}) = \underline{t^p}_{C_1 \cup C_2}$
2. $superpose(\underline{f(\overrightarrow{a_n})^p}_{C_1}, \underline{f(\overrightarrow{b_n})^p}_{C_2}) = \underline{f(\overrightarrow{superpose(a_n, b_n)})^p}_{C_1 \cup C_2}$
3. $superpose(\underline{\underline{f(\overrightarrow{a_n})^p}}_{C_1}, \underline{\underline{f(\overrightarrow{b_n})^p}}_{C_2}) = \underline{\underline{superpose(f(\overrightarrow{a_n})^p, f(\overrightarrow{b_n})^p)}}_{C_1 \cup C_2}$
4. $superpose(t_1^p, t_2^p) = \underline{erase(t_1^p)}_{\emptyset}$

Definition 9 *In the following COL is a given set of colours, $c \in COL$, $C \subseteq COL$; p_1 and p_2 are atomic polarised terms, f is a function. A,B,$\overrightarrow{A_n}$ and $\overrightarrow{B_n}$ are annotated-term variables symbol. X and $\overrightarrow{X_n}$ are term variables. The symbols $\circ$ and $\bullet$ represent compatible polarities as above. The notation $S\{p_1, \ldots, p_n\}$ is used to represent the result of appending pcdu-problems $p_1, \ldots, p_n$ at the end of sequence S. We define the transformation rules for the algorithm as follows:*

- DELETE

  $\langle \sigma, S\{\langle a^\circ, a^\bullet, A_1, A_2\rangle\}, \tau\rangle \Rightarrow$

  $\quad \langle \sigma\{\underline{a^\circ}_{\{c\}}/A_1, \underline{a^\bullet}_{\{c\}}/A_2\}, S, \tau\rangle$

  constraints: $constant(a)$, $select\_colour(c, COL)$.

- DECOMPOSE

  $\langle \sigma, S\{\langle f(\overrightarrow{a_n})^\circ, f(\overrightarrow{b_n})^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$

  $\quad \langle \sigma\{\underline{f(\overrightarrow{A_n})^\circ}_C/A, \underline{f(\overrightarrow{B_n})^\bullet}_C/B\}, S\{\overrightarrow{\langle a_n, b_n, A_n, B_n\rangle}\}, \tau\rangle$

  constraints: $C = \overrightarrow{col(A_n)}$, $C = \overrightarrow{col(B_n)}$

- ELIMINATE-L

$\langle \sigma, S\{\langle X^\circ, b^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$

$\langle \sigma\{\underline{X^\circ}_{\{c\}}/A, \underline{b^\bullet}_{\{c\}}/B\}, S, \tau\{b/X\}\rangle$

constraints: either $\{b/X\} \in \tau$ *or* $X \notin dom(\tau)$,*select_colour*(c, COL).

- ELIMINATE-R

$\langle \sigma, S\{\langle a^\circ, X^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$

$\langle \sigma\{\underline{a^\circ}_{\{c\}}/A, \underline{X^\bullet}_{\{c\}}/B\}, S, \tau\{a/X\}\rangle$

constraints: either $\{a/X\} \in \tau$ *or* $X \notin dom(\tau)$,*select_colour*(c, COL).

- IMITATE-L

$\langle \sigma, S\{\langle X^\circ, f(\overrightarrow{b_n})^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$

$\langle \sigma\{\underline{X^\circ}_C/A, \underline{f(\overrightarrow{B_n})^\bullet}_C/B\}, S\{\overrightarrow{\langle X_n, b_n, A_n, B_n\rangle}\}, \tau\{f(\overrightarrow{A_n})/X\}\rangle$

constraints: $C = \overrightarrow{col(A_n)}$, $C = \overrightarrow{col(B_n)}$ *and*

either $\{f(\overrightarrow{A_n})/X\} \in \tau$ *or* $X \notin dom(\tau)$.

- IMITATE-R

$\langle \sigma, S\{\langle f(\overrightarrow{a_n})^\circ, X^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$

$\langle \sigma\{\underline{f(\overrightarrow{A_n})^\circ}_C/A, \underline{X^\bullet}_C/B\}, S\{\overrightarrow{\langle a_n, X_n, A_n, B_n\rangle}\}, \tau\{f(\overrightarrow{B_n})/X\}\rangle$

constraints: $C = \overrightarrow{col(A_n)}$, $C = \overrightarrow{col(B_n)}$ *and*

either $\{f(\overrightarrow{B_n})/X\} \in \tau$ *or* $X \notin dom(\tau)$.

- HIDE-L

$\langle \sigma, S\{\langle f(\overrightarrow{a_n})^\circ, b, A, B\rangle\}, \tau\rangle \Rightarrow$

$\langle \sigma\{\underline{\boxed{f(\overrightarrow{A_n})^\circ}}_C/A, Sup/B\}, S\{\overrightarrow{\langle a_n, b, A_n, B_n\rangle}\}, \tau\rangle$

constraints: $Sup = superpose(B_i)$, $C = col(Sup) \neq \emptyset$

- HIDE-R

$\langle \sigma, S\{\langle a, f(\overrightarrow{b_n})^\circ, A, B\rangle\}, \tau\rangle \Rightarrow$

$\langle \sigma\{Sup/A, \underline{\boxed{f(\overrightarrow{B_n})^\circ}}_C/B\}, S\{\overrightarrow{\langle a, b_n, A_n, B_n\rangle}\}, \tau\rangle$

constraints: $Sup = superpose(A_i)$, $C = col(Sup) \neq \emptyset$

- NOMATCH $\langle \sigma, S\{\langle p_1, p_2, A, B\rangle\}, \tau\rangle \Rightarrow \langle \sigma\{\underline{p_1}_\emptyset/A, \underline{p_2}_\emptyset/B\}, S, \tau\rangle$

The function *col* returns the set of colours of an annotated term and is defined as:

$$col(\underline{t}_C) = C$$

In the *select_colour* relation, the first argument is a member of the set of colours in the second argument.

The function *dom* returns the domain of a substitution, that is, the set of variables to which terms are assigned.

6.4 Properties of PCDU

In this section we state and prove a few properties of PCDU algorithm. The way we will proceed is by first stating some lemmata needed to prove the properties of the algorithm. Next we prove the properties and, at the end, we give the proofs of the lemmata.

The following notation represents a transformation using rule R:

$$\langle \sigma, S, \tau \rangle \stackrel{R}{\Longrightarrow} \langle \sigma', S', \tau' \rangle$$

We use D for derivations, that is, rule applications chained together:

$$\langle \sigma, S, \tau \rangle \stackrel{D}{\Longrightarrow} \langle \sigma', S', \tau' \rangle$$

In this case we assume that derivation D consists of the transformations:

$$\langle \sigma, S, \tau \rangle \stackrel{R_1}{\Longrightarrow} \langle \sigma_1, S_1, \tau_1 \rangle \stackrel{R_2}{\Longrightarrow} \cdots \stackrel{R_n}{\Longrightarrow} \langle \sigma_n, S_n, \tau_n \rangle \stackrel{R_{n+1}}{\Longrightarrow} \langle \sigma', S', \tau' \rangle$$

and we use: $\langle \sigma_i, S_i, \tau_i \rangle \in D$ and $R_i \in D$ to refer to triples and rules occurring in derivation D.

When referring to a triple $\langle \sigma_i, S_i, \tau_i \rangle$ we use a_i,b_i,A_i and B_i to talk about the elements of S_i.

All the rules in the algorithm have the form:

$$\langle \sigma, S\{q\}, \tau \rangle \Longrightarrow \langle \sigma\rho, ST, \tau\kappa \rangle$$

ρ,κ and T may be empty. $\sigma\rho$ and $\tau\kappa$ represent substitution composition while ST represents concatenation of sequences. We call T the *subproblems* of q in the rule.

Lemma 3 *All rules in PCDU introduce well-formed polarised coloured annotated terms* (pcats).

Lemma 4 *For any pcdu-problem with non-empty set of terms, there is at least one rule from PCDU applicable to it.*

Lemma 5 *Let* $\langle \sigma, S\{\langle a, b, A, B\rangle\}, \tau\rangle \stackrel{R}{\Longrightarrow} \langle \sigma', ST, \tau'\rangle$ *be a transformation and* $\hat{\sigma}$ *a variable substitution. Then, rule R introduces well-formed pcats:*

If

$$\forall \langle a', b', A', B'\rangle \in T. \exists C \subseteq COL. A'\sigma'\hat{\sigma} \in AT_C \text{ and } B'\sigma'\hat{\sigma} \in AT_C$$

then

$$\exists C \subseteq COL. A\sigma'\hat{\sigma} \in AT_C \text{ and } B\sigma'\hat{\sigma} \in AT_C$$

Lemma 6 *Let* $\langle \sigma, S\{\langle a, b, A, B\rangle\}, \tau\rangle \stackrel{R}{\Longrightarrow} \langle \sigma', ST, \tau'\rangle$ *be a transformation and* $\hat{\sigma}$, $\hat{\tau}$ *variable substitutions. Then, rule R is stepwise skeleton preserving modulo polarity:*

If

$$\forall c \in COL. \forall \langle a', b', A', B'\rangle \in T. skel(A'\hat{\sigma}, c)\hat{\tau} \rightleftharpoons skel(B'\hat{\sigma}, c)\hat{\tau}$$

then

$$\forall c \in COL. skel(A\sigma'\hat{\sigma}, c)\tau'\hat{\tau} \rightleftharpoons skel(B\sigma'\hat{\sigma}, c)\tau'\hat{\tau}$$

Property 6 *Rules are erasure preserving. That is, if R is a rule such that:*

$$\langle \sigma, S\{\langle a, b, A, B\rangle\}, \tau\rangle \stackrel{R}{\Longrightarrow} \langle \sigma', ST, \tau'\rangle$$

- $\forall \langle a', b', A', B'\rangle \in T. erasure(a') = a$
- $\forall \langle a', b', A', B'\rangle \in T. erasure(b') = b$

Proof: This is easily verifiable by examining the rules of the algorithm.□

Property 7 *PCDU is terminating. If rules PCDU are applied exhaustively to a triple* $\langle \{\}, \{\langle t_1, t_2, At_1, At_1\rangle\}, \{\}\rangle$, *the process will always terminate and the outcome will be a triple of the form:* $\langle \sigma, \{\}, \tau\rangle$.

Proof: The algorithm is terminating because in every transformation:

$$\langle \sigma, S\langle a, b, A, B\rangle, \tau\rangle \stackrel{R}{\Longrightarrow} \langle \sigma', ST, \tau'\rangle$$

one of the two following cases holds:

- $T = \{\}$.
- $\forall \langle a', b', A', B' \rangle \in T$ either $a' < a$ and $b' \leq b$ or $b' < b$ and $a' \leq a$.

The $<$ relation is the subterm relation. This means that after every rule application, either the set of problems is reduced or the size of at least one of the terms in every newly introduced problem is reduced. Since $<$ is well-founded, using the multiset ordering [Jouannaud *et al* 82] on the sequences of pcdu-problems, we can conclude that the algorithm terminates.

From Lemma 4 we conclude that the final triple must have an empty set of terms. □

Property 8 *PCDU is correct. Let* $\langle \{\}, \{\langle a, b, A, B \rangle\}, \{\} \rangle \stackrel{D}{\Longrightarrow} \langle \sigma, \{\}, \tau \rangle$ *be a derivation. Then, for some* $C \subseteq COL$,

$$A\sigma\tau \in AT_C/a \text{ and } B\sigma\tau \in AT_C/b$$
$$\forall c \in C. skel(A\sigma, c)\tau \rightleftharpoons skel(B\sigma, c)\tau$$

Proof: We prove that if D is a derivation such that:

$$\langle \sigma, S, \tau \rangle \stackrel{D}{\Longrightarrow} \langle \sigma', \{\}, \tau' \rangle$$

then for all $\langle a, b, A, B \rangle \in S$, and some $C \subseteq COL$:

$$A\sigma \in AT_C/a \text{ and } B\sigma \in AT_C/b. \tag{6.1}$$

$$\forall c \in C. skel(A\sigma, c)\tau \rightleftharpoons skel(B\sigma, c)\tau \tag{6.2}$$

We do induction on the size of D. We know from Lemma 4 that D has at least one rule, so our base case is when $|D| = 1$. In this case, the preconditions of Lemmas 5 and 6 are satisfied because the set of subproblems of any $\langle a, b, A, B \rangle$ is empty and since D is just one rule we conclude from those lemmas that the theorem holds.

For the step case we assume that 6.1 and 6.2 hold when $|D| \leq m$. We now prove it for $|D| = m+1$. Let's assume that $|D| = m+1$ and let R be the first rule in D such that:

$$\langle \sigma, S, \tau \rangle \stackrel{R}{\Longrightarrow} \langle \sigma'', S'', \tau'' \rangle \stackrel{D'}{\Longrightarrow} \langle \sigma', \{\}, \tau' \rangle$$

Now, since $|D'| = m$, from the induction hypothesis we have that for all $\langle a'', b'', A'', B'' \rangle \in S''$ and some $C \subseteq COL$:

$$A''\sigma \in AT_C/a'' \text{ and } B''\sigma \in AT_C/b''.$$

$$\forall c \in C.skel(A''\sigma, c)\tau \rightleftharpoons skel(B''\sigma, c)\tau$$

and from Lemmas 5, 6 and 6 we conclude that Expressions 6.1 and 6.2 hold.

□

Proof of Lemma 4. Rule *NOMATCH* applies to any triple with a non empty problem sequence.□

Proof of Lemma 5. Let $\langle \sigma, S\{\langle a, b, A, B \rangle\}, \tau \rangle \xRightarrow{R} \langle \sigma', ST, \tau' \rangle$ be a transformation.

In order to prove this lemma, it is useful to notice the following properties of the rules:

- $\exists a'.a'/A \in \sigma'$
- $\exists b'.b'/B \in \sigma'$
- $\forall v.[v \in var(a') \leftrightarrow \exists \langle a', b', A', B' \rangle \in S'.v = A']$
- $\forall v.[v \in var(b') \leftrightarrow \exists \langle a', b', A', B' \rangle \in S'.v = B']$
- $\forall z.[z \in arg(a) \leftrightarrow \exists \langle a', b', A', B' \rangle \in S'.z = a']$
- $\forall z.[z \in arg(b) \leftrightarrow \exists \langle a', b', A', B' \rangle \in S'.z = b']$

According to these, the terms to be instantiated in A and B are constructed in a top-down "telescopic" way by introducing partial terms in σ' whose arguments are part of new problems in S'. These partial terms are annotated terms whose erasures may be constants (when S' is empty, i.e. in rules *DELETE* and *NOMATCH*), functions applied to variables to be instantiated in new problems or plain variables. The variables in the latter case are instantiated by substitution τ' which always substitutes ground terms for variables. All annotated partial terms of this last kind (with a variable as erasure) in σ become well-formed annotated terms, according to Definition 2, when τ' is applied to them.

Let us assume that:

$$\forall c \in COL. \forall \langle a', b', A', B' \rangle \in S'. skel(A_i\hat{\sigma}, c)\hat{\tau} \rightleftharpoons skel(B_i\hat{\sigma}, c)\hat{\tau} \tag{6.3}$$

we now prove by casesplit on rule R that

$$\exists C \subseteq COL. A\sigma'\hat{\sigma} \in AT_C \text{ and } B\sigma'\hat{\sigma} \in AT_C \tag{6.4}$$

Now, if R is *DELETE*, it is clear that the terms instantiated into A and B in σ' are well-formed so 6.4 holds for $C = \{c\}$.

In *DECOMPOSE*, Since all A_i and B_i are well-formed and the constraints of the rule guarantee that the set of colours C for A and B is the same as that of their arguments, we have 6.4 is true.

In *ELIMINATE-R* and *ELIMINATE-L*, a constant is unified with a variable. This is done by forming the annotated term for the variable in σ' and leaving the final instantiation of the variable (with the constant) as part of τ'. This is done like this to ensure that the terms instantiated in the variables of the original terms are not annotated –see definition 7. Now since S' is empty, by making $\hat{\sigma} = \tau'$ and $C = \{c\}$ we have that 6.4 holds.

In *IMITATE-R* and *IMITATE-L*, a function term is unified with a variable. These rules are similar to the previous ones except that the function is iterated over new variables to be instantiated in the new problems. From assumption 6.3 we know they are well-formed and the constraints ensure that the set of colours is the same so 6.4 is true.

HIDE-L and *HIDE-R* are a bit more complicated. There, a function term is hidden by a wave front in the term instantiated in either A or B. The term instantiated in the other variable (either A or B), is the result of superposing a set of annotated terms. From our assumption (6.3), we know the terms being superposed are well formed. It is easy to verify with Definitions 2 and 8 that the superposition of terms is well-formed. Since the constraints of the rule ensure that C is the set of colours of the superposed term, we have that 6.4 holds. □

Proof of Lemma 6. Let $\langle \sigma, S\{\langle a, b, A, B \rangle\}, \tau \rangle \stackrel{R}{\Longrightarrow} \langle \sigma', SS', \tau' \rangle$ be a transformation.

For the proof of this Lemma, the arguments are similar to those in the proof of Lemma 5. Let us assume that:

$$\exists\hat{\sigma},\hat{\tau}\forall c \in COL.\forall\langle a',b',A',B'\rangle \in S'.skel(A_i\hat{\sigma},c)\hat{\tau} \rightleftharpoons skel(B_i\hat{\sigma},c)\hat{\tau}$$

σ' always introduces terms compatible modulo polarity. In *DELETE*, they are the same constant with opposite polarity and the same set of colours.

In *DECOMPOSE*, if the arguments of the function are compatible modulo polarity —which according to our assumption they are—, we can see from Definitions 5 and 3 that:

$$\forall c \in COL.skel(A\sigma'\hat{\sigma},c)\tau'\hat{\tau} \rightleftharpoons skel(B\sigma'\hat{\sigma},c)\tau'\hat{\tau}$$

The components of the rules $ELIMINATE$ and the rules $IMITATE$ are similar. In the rules for $HIDE$, however, we have again superposition of terms. When this rules are applied, wave fronts are introduced in σ' both in the superposed term and the counterpart. This means that the skeleton of $A\sigma\hat{\sigma}$ and $B\sigma\hat{\sigma}$ will depend on the skeletons of their arguments according to Definition 3. But, since from our assumption the arguments are compatible modulo polarity, from Definition 6 we can conclude that 6.4 holds.□

6.5 Summary

In this chapter we have defined important notions related to the version of difference unification that we use in this thesis. We defined polarised annotated terms where both polarity and wave annotations are combined.

This notions are combined in the definition of pcd-unifiability, when two terms are unifiable by variable substitution, terms hiding as well as polarity compatibility. PCDU algorithm, also introduced in this chapter, is a correct and terminating algorithm for this notion of unification.

Chapter 7

Polarised Term Rewriting

In Chapter 4, we saw an example of how polarity and wave annotations can be used to guide the application of rewrite rules. The rewrite rules were said to have been extracted from a signature by interpreting inference rules as proof tools. In this chapter we will look more closely at how this process works.

7.1 Polarised Rewrite Rules

Rules in logic presentations express a logical relationship between judgements rather than the way in which those rules can be used to prove a theorem. In the work presented here, however, we are more interested in exploiting the practical proof capabilities of those rules; we want to have a set of sound tools to prove theorems in a particular logic. Before a proof attempt is started, a set of rewrites is extracted from the logic presentation corresponding to the theorem. The rewrite rules extracted correspond to various ways in which the inference rules can be applied in a proof and are intended to be candidates to become wave rules. Here we only look at rewrite and wave rules extracted from the signature. We mentioned in Chapter 4 the possibility of using rewrite and wave rules obtained from sources other than inference rules (e.g. lemmas proven by the user); similar techniques would be applicable in those cases.

In Section 4.2.3 we discussed a procedure to extract rewrite rules from a signature. We mentioned various non-standard rewrites rules that are used to reflect the way inference rules are used in the proofs. We now revisit these and explain informally how

they are used. In the rest of this chapter we formalise these notions.

Twin rules are rules that incorporate into rippling the effect of a branching proof. In LF, branching is produced when a curried rule is used to refine the goal. In other type theories (cf Section 2.1), a branching proof is usually represented using the product type constructor. In LF there is no type constructor to represent conjunction of proofs so we use twin rules to represent this effect at the planning level.

An example of a branching rule is rule $\wedge_i$ from the same (Natural Deduction style) presentation of Propositional Logic:

$$\Pi_{A,B:o} true(A) \rightarrow true(B) \rightarrow true(A \wedge B)$$

This rule can be used to refine a goal whose main connective is a conjunction. The result of rewriting will be two sequents each having one of the conjuncts as goal. The extracted rewrite for this inference rule is a rule whose right-hand side is a set of expressions:

$$true(A^+ \wedge B^+)^+ \Longrightarrow \left\{ \begin{array}{l} true(A^+)^+ \\ true(B^+)^+ \end{array} \right. \tag{7.1}$$

When this rewrite rule is applied, the subexpression rewritten has two simultaneous rewritings: the application of the rule will split the proof in two cases. The interpreter of the rule creates two sequents; each one has the subexpression matching the left hand side of the rule rewritten with one of the right hand side. For example, if we have a sequent of the form:

$$\Gamma \vdash true(r)^- \rightarrow true(p \wedge q)^+$$

the application of rule 7.1 would produce the following new subgoals:

$$\Gamma \vdash true(r)^- \rightarrow true(p)^+$$

$$\Gamma \vdash true(r)^- \rightarrow true(q)^+$$

In a type theory with product types, the rule for $\wedge_i$ can be expressed using that constructor:

$$\Pi_{A,B:o} true(A) \times true(B) \rightarrow true(A \wedge B)$$

In this case, the corresponding rewrite rule does not need to be a twin rule:

$$true(A^+ \wedge B^+)^+ \Longrightarrow true(A^+)^+ \times true(B^+)^+$$

In the procedure to extract rewrite rules from a signature we discussed in Section 4.2.3, there is a step where rules may be simplified. In our work we have identified one useful simplification, for the logics in Natural Deduction style we have used for experimentation, but there may be more such simplifications for other logic presentations. In Section 4.2.3 we saw the rule $\vee_e$ as an example:

$$\frac{\begin{array}{ccc} & [A] & [B] \\ & \vdots & \vdots \\ A \vee B & C & C \end{array}}{C}$$

is encoded as:

$$\Pi_{A,B,C:o} true(A \vee B) \rightarrow (true(A) \rightarrow true(C)) \rightarrow (true(B) \rightarrow true(C)) \rightarrow true(C)$$

This rule is used to eliminate a disjunction in the context. C is simply a place holder for the goal which will remain unchanged after the application of the rule. For instance, applied backwards to a sequent like:

$$\Gamma, true(p \vee q) \vdash true(r)$$

and matching $true(A \vee B)$ with $true(p \vee q)$, $\vee_e$ would produce two sequents:

$$\Gamma, true(p \vee q) \vdash (true(p) \rightarrow true(r))$$

$$\Gamma, true(p \vee q) \vdash (true(q) \rightarrow true(r))$$

if $\rightarrow$-right rule in applied to the goal to introduce $\rightarrow$, in both sequents, we obtain:

$$\Gamma, true(p \vee q), true(p) \vdash true(r)$$

$$\Gamma, true(p \vee q), true(q) \vdash true(r)$$

This set of operations to refine $true(p \vee q)$ with rule $\vee_e$ and then normalise the result by introducing $\rightarrow$ is combined, in our system, to form a tactic. We deem this tactic's behaviour as rewriting hypothesis $true(p \vee q)$ into $true(p)$ and $true(q)$ simultaneously.

Rules of this form represent a special case; to extract rewrite rules from them, the system recognises that the goal does not change and it is only the hypothesis $A \vee B$ which is rewritten in two different ways:

$$true(A^- \vee B^-)^- \Longrightarrow \begin{cases} true(A^-)^- \\ true(B^-)^- \end{cases}$$

These examples motivate the definition of the set of rewrite rules corresponding to a signature presented in the following section.

7.2 The Set of Rewrite Rules of a Signature

In this section we will define the set of polarised rewrite rules, $RW(\Sigma)$, that correspond to a signature Σ. In this set we incorporate the simplification of the of rewrites as well as the non-standard rewrites we mention in the previous section and in Section 4.2.3

We first define some auxiliary functions that will help us define the set $RW(\Sigma)$ below.

We use the function $pol(term, \Sigma, p)$ to denote the term annotated with polarity at the object level with respect to signature Σ starting with polarity p at the top level as described in Chapter 5.

Definition 10 *Given an inference rule r in a signature, we define the following functions:*

1.
   - $head(j) = j$ *if* j *is an atomic judgement.*
   - $head(j_1 \rightarrow j_2) = head(j_2)$
   - $head(\Pi_{a:o} j) = head(j[X/a])$ *where* X *is a fresh meta variable.*

2.
   - $antec(j) = []$ *if* j *is an atomic judgement.*
   - $antec(j_1 \rightarrow j_2) = [j_1 | antec(j_2)]$
   - $antec(\Pi_{a:o} j) = antec(j[X/a])$ *where* X *is a fresh meta variable.*

3.
   - $drop(j, j_1) = j_1$ *if* j_1 *is an atomic judgement.*
   - $drop(j, j_1 \rightarrow j_2) = j_2$ *if* $unifiable(j, j_1)$.

- $drop(j, j_1 \rightarrow j_2) = j_1 \rightarrow drop(j, j_2)$ *if* $\neg unifiable(j, j_1)$.
- $drop(j, \Pi_{a:o} j_1) = drop(j, j_1[X/a])$ *where* X *is a fresh meta variable.*

4.
- $collapse(j) = \{j_1, \ldots, j_n\}$ *if* $j = (j_1 \rightarrow k) \rightarrow \ldots (j_n \rightarrow k) \rightarrow k$.
- $collapse(j) = \{j\}$ *otherwise.*

Head returns the head of an LF rule and *antec* returns a list with the judgements in the body an LF rule as defined in Section 2.1.1. *Drop* removes a given judgement from the body of the rule. Finally, *collapse* removes dummy variables (as in mentioned in Section 7.1 for rule $\vee_e$) from a rule and returns the list of judgements to become the conclusion of a twin rule.

We are now finally in a position to define the set of polarised rewrite rules for a signature.

Definition 11 *Let* Σ *be a signature. If* $r \in \Sigma$, *then the following rules are in* $RW(\Sigma)$:

1.
$$pol(head(r), \Sigma, +) \Rightarrow \begin{cases} pol(a_1, \Sigma, +) \\ \vdots \\ pol(a_n, \Sigma, +) \end{cases}$$
where $a_i \in antec(r)$ *and* $\neg unconstrained(head(r))$.

2.
$$pol(a, \Sigma, -) \Rightarrow \begin{cases} pol(a_1, \Sigma, -) \\ \vdots \\ pol(a_n, \Sigma, -) \end{cases}$$
where $a \in antec(r)$,$\neg unconstrained(a)$ *and* $\{a_i\} = collapse(drop(a, r))$.

$unconstrained(t)$ ensures that t is not unifiable to any atomic judgements as explained in Section 4.2.3.

The first of the two definitions of rules above specifies right-to-left rules, that is, rules corresponding to backward-chaining. In these rules, the head of the inference rule is assigned positive polarity and polarised with respect to the signature. The right-hand side of the rewrite is formed by all the judgements in the body of the original inference rule. If there are more than one of these, the rule becomes a twin rule.

The second definition specifies left-to-right rules or rules corresponding to forward-chaining of the original inference rule. Here, for each non unconstrained judgement (as defined in Section 4.2.3) in the body of the original rule, a rewrite is produced. A non unconstrained judgement in the body is assigned negative polarity and then polarised at the object level with respect to the signature; this constitutes the left-hand side of the rewrite. The right-hand side is formed by first removing from the original inference rule the the judgement chosen to be the left-hand side of the rewrite. After this, function *collapse* simplifies the dummy variables as explained in Section 7.1.

If any of a_i contains a variable which does not occur in the left-hand side of the rule, the rule will be improper.

$RW(\Sigma)$ represents the set of rules that can be obtained using the informal procedure introduced in Section 4.2.3. The following section describes a procedure to obtain wave rules from the set $RW(\Sigma)$.

7.3 Polarised Wave Rule Parsing

Polarity annotations, as well as wave annotations, are meta level syntax used to highlight different aspects of expressions. We use both kinds of annotation when rippling. Polarity is used to guarantee that inference rules, in the form of rewrites, are applied in the correct direction. Wave annotations trace the *movement* of wave fronts and skeletons through rewriting; they restrict the application of wave rules to those which guarantee that the desired structures will be moved.

Below we present an algorithm to parse wave rules from rewrite rules. First, we define an auxiliary function that combines both types of annotation.

Definition 12 *Let p be a polarised term and a an annotated term whose erasure is $unpol(p)$ (the unpolarised version of p). Then we define $combine(p, a)$ by cases as follows:*

1. $combine(X^\bullet, \underline{X}_C) = \underline{X^\bullet}_C$
2. $combine(t^\bullet, \underline{t}_C) = \underline{t^\bullet}_C$

3. $combine(f(\overrightarrow{t_n})^\bullet, \underline{f(\overrightarrow{at_n})}_C) = \underline{f(\overrightarrow{combine(t_n, at_n)})^\bullet}_C$

4. $combine(f(\overrightarrow{t_n})^\bullet, \underline{\boxed{f(\overrightarrow{at_n})}}_C) = \underline{\boxed{f(\overrightarrow{combine(t_n, at_n)})^\bullet}}_C$

where $\bullet$ *stands for any of the polarity values* $\{+,-,\pm\}$.

In the following we will assume that if t is a term, at is an annotated version of the term, pt is a polarised version of the term and pat is a polarised annotated version of it.

Definition 13 *Let* Σ *be a signature and* $pl \Rightarrow \begin{cases} pr_1 \\ \vdots \\ pr_n \end{cases}$ *a member of* $RW(\Sigma)$. *A polarised wave rule is obtained from this rule by the following steps:*

1. *Let* $l = unpol(pl)$ *and* $r_i = unpol(pr_i)$ *for all* i
2. *D-unify* l *and* r_i *for all* i *with frozen variables (variables considered as constants). Let's call the annotated* l_i, al_i *and the annotated* r_i, ar_i.
3. *Let* $pal = combine(pl, superpose(al_i))$ *and* $par_i = combine(pr_i, ar_i)$ *for all* i.
4. *Convert colour sets in* pal *and* par_i *into different colour set variables.*
5. *The resulting polarised wave rule is* $pal \Rightarrow \begin{cases} par_1 \\ \vdots \\ par_n \end{cases}$.

In Step 1, a version of the rule elements with no polarity is generated. Then, the (unpolarised) left-hand side is d-unified with each of the (unpolarised) right-hand sides. This generates pairs of annotated lh-sides and rh-sides. In step 3, the original polarity annotations are restored onto the annotated rule through function *combine*. In Step 4, the colour sets of the terms that take part in the wave rule are changed into colour set variables. These variables match colour sets in the terms being rewritten.

7.4 Example

We now show an example of how the process works. The following rule in Σ, corresponding to $\wedge_i$ in a Propositional Logic signature:

$$\Pi_{A,B:o} true(A) \rightarrow true(B) \rightarrow true(A \wedge B)$$

gives the following member of $RW(\Sigma)$:

$$true(A^+ \wedge B^+)^+ \Rightarrow \left\{ \begin{array}{l} true(A)^+ \\ true(B)^+ \end{array} \right. \tag{7.2}$$

The unpolarised version of the rule (Step 1 in Definition 13) is:

$$true(A \wedge B) \Rightarrow \left\{ \begin{array}{l} true(A) \\ true(B) \end{array} \right.$$

Difference unifying the left-hand side with both right-hand sides (Step 2) gives the following pairs:

$$true(\underline{\underline{\boxed{\underline{A}_{\{c_1\}} \wedge \underline{B}_{\emptyset}}}}_{\{c_1\}}), \underline{true(\underline{A}_{\{c_1\}})}_{\{c_1\}}$$

$$true(\underline{\underline{\boxed{\underline{A}_{\emptyset} \wedge \underline{B}_{\{c_2\}}}}}_{\{c_2\}}), \underline{true(\underline{B}_{\{c_2\}})}_{\{c_2\}}$$

In step 3 we combine the two annotations superposing all left-hand sides:

$$true(\underline{\underline{\boxed{\underline{A}_{\{c_1\}} \wedge \underline{B}_{\{c_2\}}}}}_{\{c_1,c_2\}})^+ \Rightarrow \left\{ \begin{array}{l} \underline{true(\underline{A}_{\{c_1\}})}_{\{c_1\}} \\ \underline{true(\underline{B}_{\{c_2\}})}_{\{c_2\}} \end{array} \right.$$

In Step 4 colour sets become colour set variables:

$$true(\underline{\underline{\boxed{\underline{A}_{C_1} \wedge \underline{B}_{C_2}}}}_{C_3})^+ \Rightarrow \left\{ \begin{array}{l} \underline{true(\underline{A}_{C_1})}_{C_1} \\ \underline{true(\underline{B}_{C_2})}_{C_2} \end{array} \right.$$

The final step annotates the object-level part of the skeleton:

$$true(\underline{\underline{\boxed{\underline{A^+}_{C_1} \wedge \underline{B^+}_{C_2}}}}_{C_3})^+ \Rightarrow \left\{ \begin{array}{l} \underline{true(\underline{A^+}_{C_1})^+}_{C_1} \\ \underline{true(\underline{B^+}_{C_2})^+}_{C_2} \end{array} \right.$$

The application of wave rules to annotated terms in a sequent will produce several new sequents. Only one of them will correspond to rippling and the rest will be simple rewritings. This will be more clear after we have defined polarised term rewriting and given an example in the next section.

7.5 Polarised Annotated Term Rewriting

In this section we define what it means to apply a wave rule to a subexpression of a term in an LF sequent.

When rewriting an occurrence of an annotated term in some hypothesis, the resulting sequents have new hypotheses corresponding to different versions of the rewritten hypothesis with the term occurrence replaced by one of the consequences of the rule. The underlined polarised annotated term in $H[\underline{pat}]$ indicates an occurrence of pat in H, so the notation $H[r_i\tau/\underline{pat}]$ indicates the replacement of the same occurrence of pat with $r_i\tau$.

When rewriting an occurrence of an annotated term in the goal of the sequent, on the other hand, the resulting sequents have the goals replaced by the new goals where the term occurrence has been replaced. This asymmetry is a consequence of the way the inference rules of a signature are handled by the framework logic.

Definition 14 *Let* $r = (l \Rightarrow \left\{ \begin{array}{l} r_1 \\ \vdots \\ r_n \end{array} \right.)$ *be a polarised wave rule as described in last section. If* $l\tau = pat$ *for some capture-avoiding substitution* τ*, then:*

- $\Gamma_1, H[\underline{pat}], \Gamma_2 \vdash C$ *rewrites by* r *to:*

$$\begin{array}{c} \Gamma_1, H, \Gamma_2, H[r_1\tau/\underline{pat}] \vdash C \\ \vdots \\ \Gamma_1, H, \Gamma_2, H[r_n\tau/\underline{pat}] \vdash C \end{array}$$

- $\Gamma \vdash C[pat]$ *rewrites by* r *to:*

$$\begin{array}{c} \Gamma \vdash C[r_1\tau/\underline{pat}] \\ \vdots \\ \Gamma \vdash C[r_n\tau/\underline{pat}] \end{array}$$

These two cases for rewriting hypotheses and the goal define a rewriting relationship over annotated terms. The set of polarised annotated terms is closed under this relationship but for this to be true, there are restrictions on the substitution of variables τ.

In order to ensure well-formedness under substitution of annotated terms, in [Basin & Walsh 93], substitution is modified as follows: all annotation is removed from an annotated term substituted for all occurrences of a variable in the wave front of a term; annotation is preserved when substituting for variable occurrences in a wave hole. This restriction is also needed for the termination measure for rippling presented there. Here, we use the same restriction.

Definition 14 assumes that rules are first order. The problem of higher-order rippling is hard and it is currently under research [Smaill & Green 96],[Hutter & Kohlhase 95]. We don't address this problem in this thesis.

Let us now see an example of annotated rewriting. Suppose we have the following sequent:

$$\begin{array}{rcl} hyp_1 & : & true(\boxed{\underline{(a^-_{\{c_1\}} \wedge b^-_{\{c_2\}})^-}_{\{c_1,c_2\}} \vee \underline{c^-}_{\emptyset}})^- \\ & \vdash & true(\boxed{\underline{\boxed{(a^+_{\{c_1\}} \vee c^+_{\emptyset})^+}}_{\{c_1\}}} \wedge \boxed{\underline{\boxed{(b^+_{\{c_2\}} \vee c^+_{\emptyset})^+}}_{\{c_2\}}})^+ \end{array}$$

Using wave rule 7.4 to rewrite the goal of this sequent produces the following two sequents:

$$\begin{array}{rcl} hyp_1 & : & true(\boxed{\underline{(a^-_{\{c_1\}} \wedge b^-_{\{c_2\}})^-}_{\{c_1,c_2\}} \vee \underline{c^-}_{\emptyset}})^- \\ & \vdash & true(\underline{\boxed{(a^+_{\{c_1\}} \vee c^+_{\emptyset})^+}}_{\{c_1\}})^+ \end{array}$$

$$\begin{array}{rcl} hyp_1 & : & true(\boxed{\underline{(a^-_{\{c_1\}} \wedge b^-_{\{c_2\}})^-}_{\{c_1,c_2\}} \vee \underline{c^-}_{\emptyset}})^- \\ & \vdash & true(\underline{\boxed{(b^+_{\{c_2\}} \vee c^+_{\emptyset})^+}}_{\{c_2\}})^+ \end{array}$$

7.6 Summary

In this chapter we have defined the set $RW(\Sigma)$ of rewrite rules extracted from inference rules in signature Σ. We also defined an algorithm to parse the elements of this set into polarised wave rules. The meaning of polarised annotated term rewriting was also defined formally and an example was given.

Chapter 8

System Specification

This chapter describes a specification of methods and submethods that realise the search techniques presented in Chapter 4.

In Chapter 9, the behaviour of the specification and an implementation of an earlier version will be discussed. We first give an overview of the whole design, then we describe methods and submethods.

The methods and submethods are specifications of tactics implemented in a version of LF in Mollusc (Section 2.2). An implementation of an earlier version of the methods will be presented in Section 9.7.

8.1 Overview

In figure 8.1 we can see an overall picture of the system. The bottom right box represents Mollusc, it has LF implemented and its library contains a set of tactics specific to this framework logic. Mollusc can call the planner via the standard Mollusc interface which performs some preprocessing on the sequent to be passed to the planner as well as some postprocessing to extract the tactic from the plan after it has been completed. The preprocessing involves creating a data base of rewrite and wave rules extracted from the inference rules in the signature as described in Chapter 7.

Once the planner has built a proof plan for the sequent, the plan can be sent to Mollusc to be executed. The plan will go through the interface and from it a tactic will be handed to Mollusc. Mollusc will then execute the tactic and obtain a proof of the

Proof planning in LF.
System Schema.

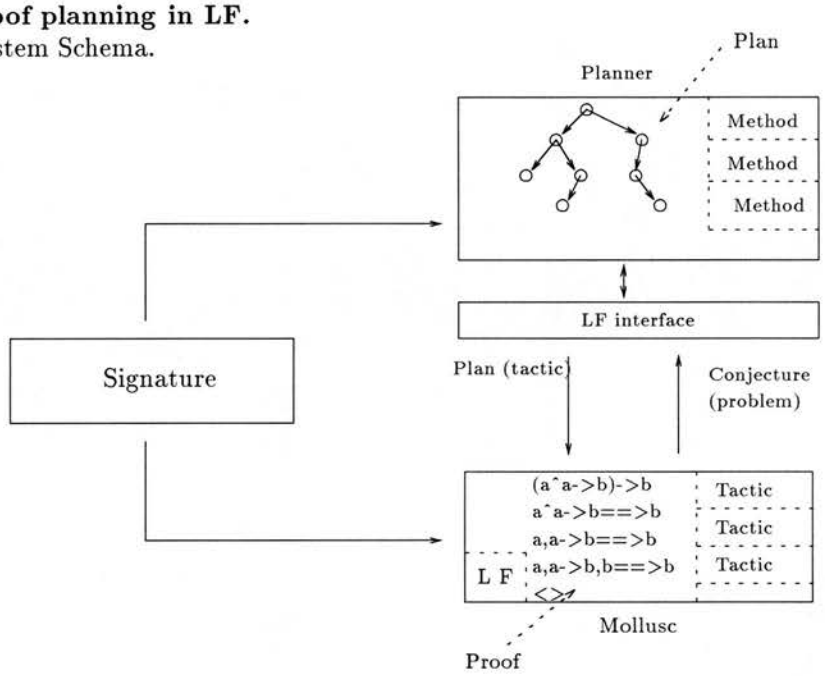


Figure 8.1: Overall system picture.

original sequent. This last step is important because it is only the proof generated by the proof editor that guarantees the soundness of the whole proof process.

On the left hand side of the picture we see arrows going from the signature to both the planner and Mollusc. These indicate that signatures parameterise both Mollusc and the planner: Mollusc's tactics use the signature to build the proof and the planner takes signatures as the source of the rewrite rules it uses to build proof plans.

8.2 Organisation of Methods and Submethods

The methods and submethods we discuss here are specifications of heuristics for the application of tactics in the Mollusc implementation of LF. As such, they are not complete in the sense of always being able to find a proof whenever there exists one (cf Section 4.4).

There are five methods: `balance`, `d-reduction`, `weak-fertilise`, `axiom` and `unblock`. The general strategy is the following: `balance` applies LF and object level introduction rules to provide the sequent with some hypotheses. `D-reduction` identifies possible

connections between the goal and the hypotheses, selects one hypothesis as the closest to the goal and ripples both the selected hypothesis and the goal. If both expressions are fully rippled, either `axiom` or `weak-fertilise` will be applicable. The first one will close the branch of the plan. The second method refines the goal and produces new goals to which the whole process will be applied. If the rippling process gets stuck, method `unblock` applies rewrite rules that enable the general process to continue. In Figure 8.2 there flow diagram of the application of methods in our system.

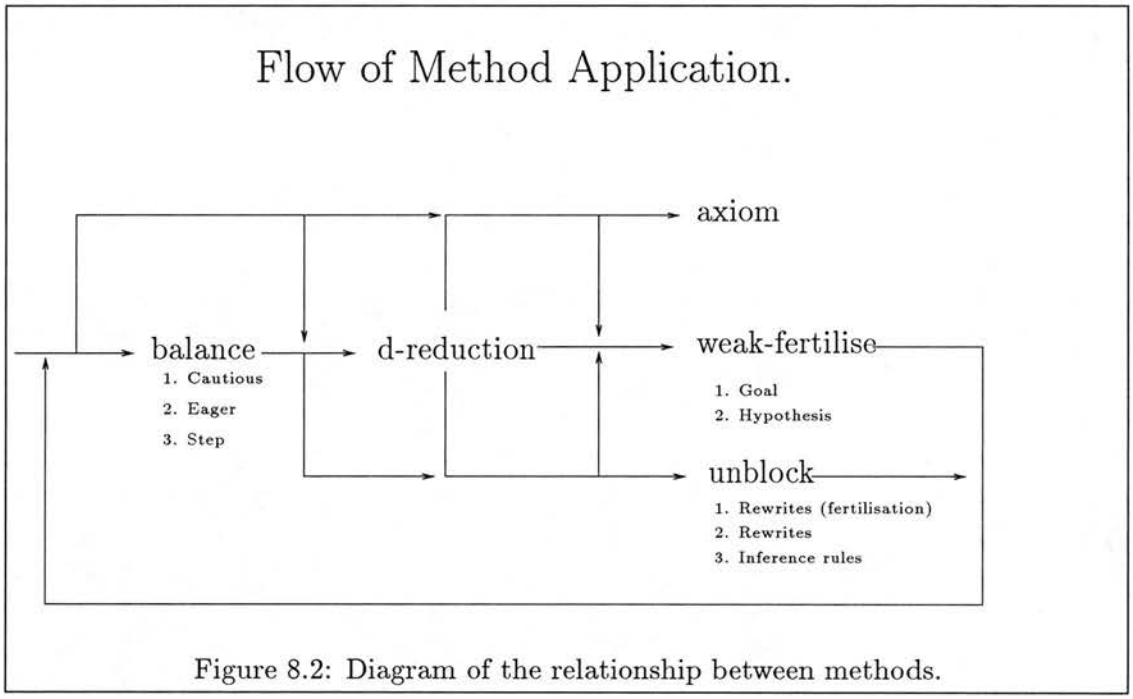


Figure 8.2: Diagram of the relationship between methods.

For the sake of clarity, we describe methods and submethods in a declarative way in a mixture of Prolog notation, set theoretic syntax and English. The specification however, assumes a Prolog-like control of the method and submethod evaluation (i.e. unification, backtracking, etc.).

Methods are selected by the planner in the order in which they appear in the method database. The planner has no direct access to the submethods however, they can only be called by the methods to perform independent and complementary tasks.

Below we give a description of each one of the methods and submethods of the system.

8.3 Methods

We start our description of methods with `axiom`, the only terminating method. Then we proceed in the order corresponding to the general description of the system given in the previous section.

8.3.1 Axiom

This method detects when a sequent can be transformed into an axiom. By this we mean that there is either a connection between the goal and some hypothesis or the head of the goal matches a member of its body and hence introducing some part of the body produces an axiom.

Since this is the only terminating method of our design, we don't need to apply the introduction rules necessary to obtain an axiom; we simply detect the possibility of obtaining one and leave it to the tactic to apply the rules. Since *axiom* is the only terminating method, it should be at the leaves of all proof plans.

`Method:` *axiom*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $head(EGoal, Head)$
3. $body(EGoal, Body)$
4. $member(Head, Body)$ *or* $member(Head, EHyp\_list)$

Postconditions: `None`

New goals: `None`

The preconditions of the method remove all annotation from the input sequent before checking the property stated above. $Head$ is the head of the erasure of the goal and $Body$ is the list of types in the body of the goal.

8.3.2 Balance

In the beginning of a proof, a sequent is often composed of a goal only and there are no hypotheses. Our mechanism needs hypotheses in order to work, so a degree of introduction into the hypothesis list needs to be done before difference reduction can be started. The task of the `balance` method is to do this introduction. The method is not only used in the beginning of a proof however, it is also used to balance new goals introduced by weak-fertilisation.

We have specified three different versions of the method. One version applies introduction and elimination rules from both the object logic[1] and the LF level as many times as possible. We call this kind of balance *eager* balance and it extracts the maximum number of hypotheses from a goal. By introducing as much as possible, this kind of balance introduces some connecting expressions in the hypothesis list. These connections cannot be accessed directly in our design because we use difference unification between the goal and the hypotheses to identify them. Some times `weak-fertilising` brings one of the connecting expressions from the hypothesis list back into the goal. In these cases the connections can be identified by our difference reduction mechanism but there are situations when what we need is a different kind of balance[2].

The second type of balance also applies object and LF level introduction rules but tries to maximise the number of potential connections across the sequent symbol. It does this by introducing in the hypothesis list only if at least one atomic subterm of the judgement introduced is also a subterm of the goal. For example, given the following sequent:

$$\vdash true(((a \wedge b) \supset c) \supset a \supset b \supset c)$$

Balance will find an object introduction rule applicable to the goal, that is imp_i:

$$\vdash true(((a \wedge b) \supset c)) \rightarrow true(a \supset b \supset c)$$

---

[1] An introduction rule from the object logic is a rewrite rule whose left-hand side has positive polarity. Correspondingly, an elimination rule has negative polarity in the left-hand side.

[2] See Example 3 in Chapter 9

and then apply $\rightarrow r$ to obtain a new hypothesis:

$$\begin{array}{rcl} hyp_1 & : & true(((a \wedge b) \supset c) \\ & \vdash & true(a \supset b \supset c) \end{array} \tag{8.1}$$

These steps introduce in the hypothesis list atomic subterms a, b and c which are also part of the goal. Trying again the same procedure would involve applying the same object level introduction rule *imp_i* and $\rightarrow r$. In this case the atomic subterm introduced into the hypothesis list would be a; but a, at that point, won't be part of the goal any more so the second part is not acceptable. As we can see in Sequent 8.1, the hypothesis and the goal have the same atomic subterms. This enables coloured difference-unification to find all the connections from the start without leaving connections amongst the hypotheses. We call this kind of balance *cautious.*

The third kind of balance is a complement to the other two. It simply applies introduction rules (either LF or object-level) one at a time. This version acts as a default in case the other two are not successful and gives the planner the option to continue stepwise. We call this kind of balance *step* balance.

For the specification of eager and cautious balance we use what is called in Clam (Section 3.2) an *iterator.* An iterator is a method that recursively applies a list of methods or a list of submethods to a sequent and its descendants in the tree as far as possible. Its precondition is that at least one of the given methods or submethods is applicable and the output is a list of all the sequents where the members of the given list could not be applied.

For eager balance we use an iterator over submethods `intro-LF` (Section 8.4.1), `intro` (Section 8.4.2), `elim-LF` (Section 8.4.4) and `elim` (Section 8.4.4). The first submethod applies LF introduction rules and the second applies introduction rules at the object level. Similarly, `elim-LF` applies LF elimination rules and `elim` applies object-level elimination rules.

For *cautious* balance, there is an iterator over submethods `intro-LF` and `intro-con` (*Intro connections*, Section 8.4.3) and performs the operation described above where only atomic subterms present in the goal are introduced.

The three clauses for the three versions of the *balance* method are:

`Method:` *balance*

(Clause 1:)

`Iterator of submethods intro-lf and intro-con`

(Clause 2:)

`Iterator of submethods intro-lf, intro, elim-lf and elim`

(Clause 3:)

Input: *Sequent*

Preconditions:

1. $applicable\_submethod(intro\_LF, Sequent, New\_goals)$ or
   $applicable\_submethod(intro, Sequent, New\_goals)$

Postconditions: `None`

New goals: `New-goals`

8.3.3 D-reduction

This method d-unifies the goal successively against all hypotheses, selects the hypothesis which is *closest* to the goal and applies reduction submethods to make the difference between the goal and selected hypothesis smaller.

`Method:` *d_reduction*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $Pairs = \{ \langle ah, ag \rangle \mid cdu(h, g, ah, ag) \wedge h \in Hyp\_list \}$
2. $sorted(Pairs, Sorted\_pairs)$
3. $selection(A\_pair, Sorted\_pairs)$
4. $annotated\_sequent(A\_pair, Hyp\_list \vdash Goal, New\_hyp \vdash New\_goal)$

5. $applicable\_submethod(reduce, New\_hyp \vdash New\_goal, New\_goals)$

Postconditions: None

New goals: New_goals

The preconditions first difference unify all hypotheses against the goal and collect the outcome in a list of annotated hypothesis-goal pairs[3]. The list of pairs is then sorted, according to a measure of disagreement, going from the most similar to the least similar pair. The measure of a pair is calculated as the maximum between the annotation-depths of both members of the pair.

The annotation-depth of term is the distance in the term-tree between the top-most wave front and the bottom-most wave hole. The motivation for this measure comes from the following observations: First, for two annotated terms to unify, all of their wave fronts must be removed. Second, the wave rules we use either remove a wave front from the term or increase the size of a wave hole in it. The measure just described is then a parameter to count the number of rippling steps needed to remove all the wave fronts and make two terms unify. In figure 8.3.3 there is an example of a rippling step using wave rule imp_i. The expressions in boxes in the term trees mark the expressions in the wave front and those underlined show the expressions in the wave hole. As the expression on the left is rewritten into the one on the right, the wave hole is augmented and the wave front is reduced; the annotation-depth goes from 2 to 1 in the process.

The pairs in $Sorted\_pairs$ are selected from the top of the list as candidates for reduction. From the pair selected, a new sequent is built where only the relevant hypothesis is annotated as well as the goal. The last precondition is the call for the application of submethod $reduce$ on the annotated sequent. The new goals will be those collected from the submethod application.

8.3.4 Weak-fertilise

This method specifies a goal refinement by a hypothesis or a forward-chaining operation between hypotheses. For the first operation, it looks for a hypothesis whose head

[3] Notice that one hypothesis and the goal may produce more than one pair when d-unified

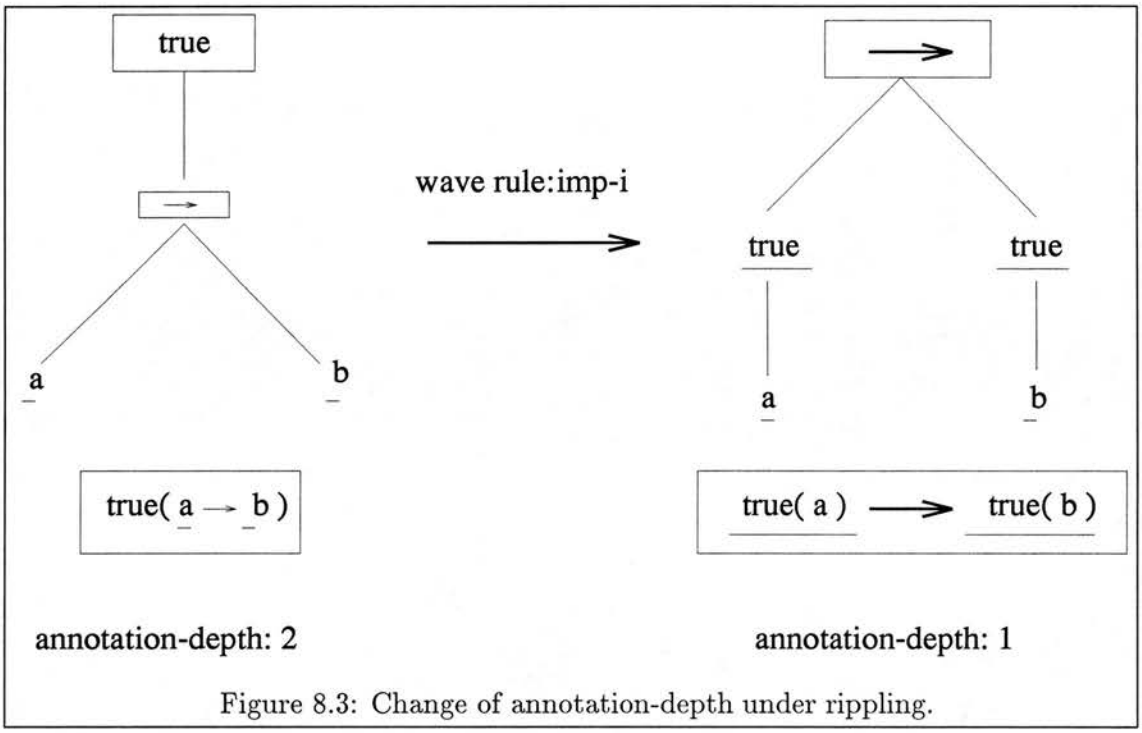


Figure 8.3: Change of annotation-depth under rippling.

matches the head of the goal and rewrites the goal with the elements of the body of the hypothesis. The new subgoals will reflect the rewriting of the goal by each one of the elements of the hypothesis' body.

The second operation consists of finding a hypothesis hyp_1 that matches an element of the body of another hypothesis hyp_2. The new sequent has a new hypothesis identical to hyp_2 but with the element of its body that matches hyp_1 removed.

In inductive proof planning, when the goal has been fully rippled and it does not match the induction hypothesis but the latter can be used to reduce the goal, we call it `weak-fertilisation`. Here we use the same terminology. The specification of the method consists of the two clauses corresponding to the two operations just described. The first clause is:

`Method:` $weak\_fertilise$

(Clause 1:)

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $head(EGoal, GoalHead)$
3. $member(Hyp, EHyp\_list)$
4. $head(Hyp, HypHead)$
5. $unify(GoalHead, HypHead)$

Postconditions:

1. $body(Goal, GoalBody)$
2. $body(Hyp, HypBody)$
3. $sequent\_update(GoalBody, EHyp\_list \vdash EGoal, NewHyp \vdash \_)$
4. $Subgoals = \{NewHyp \vdash GB_i \mid GB_i \in HypBody\}$

New goals: $Subgoals$

The preconditions state that there needs to be an unannotated hypothesis Hyp whose head unifies the head of the unannotated goal $EGoal$.

In the postconditions, the judgements of the bodies of both Hyp and $EGoal$ are put into lists ($GoalHead$ and $HypHead$). The elements of $HypBody$ will be the goals of the new sequents (GB_i) while those of $GoalBody$ will go into the hypothesis lists of the new sequents ($NewHyp$). The effect of this last operation is the introduction of the whole body of $Goal$. In the proof editor this is done by applying $\rightarrow r$ and Πr rules.

(Clause 2:)

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $member(Hyp_1, EHyp\_list)$

3. $member(Hyp_2, EHyp\_list)$

4. $body(Hyp_2, Body)$

5. $select(Jud, Body, BodyRest)$

6. $unify(Jud, Hyp_1)$

Postconditions:

1. $head(Hyp_2, Head)$

2. $make\_judgement(BodyRest, Head, NewHyp)$

New goals: $[[NewHyp|Hyp\_list] \vdash Goal]$

In this clause the preconditions find a hypothesis Hyp_1 that unifies with an element in the body of another hypothesis Hyp_2. The predicate *make_judgement* in the postconditions converts all elements of $BodyRest$ into the curried antecedents of judgement $NewHyp$ where $Head$ is the consequent. This new hypothesis is added to the input sequent to form the new $Goal$ in the New goals slot.

8.3.5 Unblock

The **unblock** method serves the purpose of applying rules which are not wave rules whenever no difference-reducing step can be taken. Unblocking in the context of inductive theorem proving has the objective of applying lemmas that allow the rippling process to continue. In our context, unblocking is more general. Whenever wave rules are not applicable, it is necessary to weaken the constraints imposed on the rules to look for ways of continuing the planning process.

The first weakening of the rules we do to drop the annotations, that is, to apply rewrite rules to try to either obtain an axiom directly or enable weak-fertilisation. The second weakening consists of allowing the application of rewrite rules without the requirement that they enable **axiom** of **weak-fertilise** to be applicable.

There are 4 clauses for this method. The first two correspond to the application of a rewrite rule to a hypothesis and to the goal respectively; the application is subject

to the applicability of `axiom` or `weak-fertilise` as explained above. The other two clauses correspond to unconstrained rewrite-rule application to hypothesis and goal respectively.

`Method:` *unblock*

(Clause 1:)

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $rewrite\_rule(RRule)$
3. $member(Hyp, EHyp\_list)$
4. $subterm(Hyp, SubHyp)$
5. $applicable\_rewrite(RRule, SubHyp, NewSubexp)$
6. $replace(SubHyp, NewSubexp, Hyp, NewHyp)$
7. $NewSequent = [NewHyp|EHyp\_list] \vdash EGoal$
8. $applicable\_method(axiom, NewSequent, \_)$ or
   $applicable\_method(weak\_fertilise, NewSequent, \_)$

Postconditions: `None`

New goals: $NewSequent$

(Clause 2:)

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $rewrite\_rule(RRule)$
3. $subterm(Goal, SubexpGoal)$

4. $applicable\_rewrite(RRule, SubexpGoal, NewSubexp)$
5. $replace(SubHyp, NewSubexp, Goal, NewGoal)$
6. $NewSequent = [NewHyp|EHyp\_list] \vdash EGoal$
7. $applicable\_method(axiom, NewSequent, \_)$ or
   $applicable\_method(weak\_fertilise, NewSequent, \_)$

Postconditions: None

New goals: $NewSequent$

(Clause 3:)

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $rewrite\_rule(RRule)$
3. $member(Hyp, EHyp\_list)$
4. $subterm(Hyp, SubHyp)$
5. $applicable\_rewrite(RRule, SubHyp, NewSubexp)$

Postconditions:

1. $replace(SubHyp, NewSubexp, Hyp, NewHyp)$
2. $NewSequent = [NewHyp|EHyp\_list] \vdash EGoal$

New goals: $NewSequent$

(Clause 4:)

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$
2. $rewrite\_rule(RRule)$
3. $subterm(SubexpGoal, Goal)$
4. $applicable\_rewrite(RRule, SubexpGoal, NewSubexps)$

Postconditions:

1. $replace(SubHyp, NewSubexp, Goal, NewGoal)$
2. $NewSequent = [NewHyp|EHyp\_list] \vdash EGoal$

New goals: $NewSequent$

8.4 Submethods

In this section we describe the submethods called by the methods from the previous section. We introduce them in the order in which they are mentioned in the previous sections.

8.4.1 Intro-LF

This submethod checks if the goal of the input sequent is a function type and modifies the sequent to reflect the application of Π or $\rightarrow$ introduction rules. It is called by the two versions of

`balance`.

`Submethod:` $intro\_LF$

(Clause 1:)

Input: $Hyp\_list \vdash A \rightarrow B$

Preconditions: `None`

Postconditions:

1. Generate a variable v fresh in Hyp_list.

New goals: $[Hyp\_list, (v : A) \vdash B]$

(Clause 2:)

Input: $Hyp\_list \vdash \Pi_{x:A} B$

Preconditions: None

Postconditions:

1. Generate a variable v fresh in Hyp_list.

New goals: $[Hyp\_list \vdash B[v/x]]$

8.4.2 Intro

Submethod `intro` is used by `balance` (eager) (Section 8.3.2). It searches the rewrite rule data base for introduction rules applicable to the goal and performs the appropriate rewriting of the goal.

Introducing at the object level is applicable when the goal is an atomic judgement; the application of an introduction rule would convert the atomic judgement into a hypothetical or conditional one. The antecedents of these last two can then be introduced at the LF level to gain hypotheses (submethod `intro-lf`).

Submethod: *intro*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $Rules = \{Lhs \Rightarrow Rhs \mid rewrite\_rule(Lhs \Rightarrow Rhs)\ and\ Lhs = Goal\}$
2. $Rules = [Lhs' \Rightarrow Rhs']$

Postconditions:

1. $Goal = Lhs'$

2. $Subgoals = \{Hyp\_list \vdash R_i \mid R_i \in Rhs'\}$

New goals: $Subgoals$

All applicable introduction rules are considered in order to verify that there is only one applicable. The reason for this is that, if there are more than one, parts of the goal may be dropped by the application of each of them. These parts cannot be recovered later in the proof and so a careful decision must be made as to which rule is the best to use. We use annotations to guide this decision in difference reduction and so in `balance` —from which `intro` is called— we stop the introduction process whenever this choice arises.

The simplest example of multiple introduction rules is a disjunction. There are two introduction rules for disjunction in Natural Deduction style Propositional logic; when proving backwards, each one of them drops a disjunct from the goal. The disjunct dropped is likely to contribute to the proof at some point so the decision must be taken when balancing, especially at the beginning of the proof.

8.4.3 Intro-con

This submethod is designed to apply introduction rules from the object level only if the body of the new term contains an atomic subterm which also occurs in the head. See Section 8.3.2 for an example of how this submethod fits in to `balance` method.

`Submethod:` $intro\_con$

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $Rules = \{Lhs \Rightarrow Rhs \mid rewrite\_rule(Lhs \Rightarrow Rhs)$ and
   $polarity(Lhs, +), applicable\_rewrite(Lhs \Rightarrow Rhs, Goal, \_)\}$

   1. $Rules = [RRule]$

   1. $applicable\_rewrite(RRule, Goal, NewGoal)$

   1. $\forall R \in NewGoal$:

(a) head(R, HeadR)

(b) body(R, BodyR)

(c) subterm(STerm, BodyR)

(d) atom(STerm)

(e) occurs(STerm, HeadR)

Postconditions:

1. $Subgoals = \{Hyp\_list \vdash RHS_i \mid RHS_i \in Rhs'\}$

New goals: *Subgoals*

The preconditions collect all the rewrite rules with positive polarity in the left-hand side that are applicable to the goal. This set must have only one element (to avoid colour dropping). After this, the new goals are computed and, for each one of them, there must be an atomic subterm of the body which occurs in the goal. See description in Section 8.3.2.

The postconditions form the new sequents with the old hypothesis list and each one of the new goals.

8.4.4 Elim-LF

Submethod `elim-lf` specifies a forward chaining operation at the LF level between hypotheses.

`Submethod:` *elim_lf*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $erasure(Hyp\_list \vdash Goal, EHyp\_list \vdash EGoal)$

2. $member(Hyp_1, EHyp\_list)$

3. $member(Hyp_2, EHyp\_list)$

4. $body(Hyp_2, Body)$

5. $select(Jud, Body, BodyRest)$

6. $unify(Jud, Hyp_1)$

Postconditions:

1. $head(Hyp_2, Head)$

2. $make\_judgement(BodyRest, Head, NewHyp)$

New goals: $[[NewHyp|Hyp\_list] \vdash Goal]$

The preconditions are that there must be a hypothesis (hyp_1) that matches a member of the body of another hypothesis (hyp_2).

The postconditions form a new hypothesis with the remaining of the of the elements of the body of hyp_2 after removing the one that matches hyp_1 and the head of hyp_2. The new sequent is the same as the old sequent but with the new hypothesis added.

8.4.5 Elim

This submethod specifies a forward chaining operation at the object level by applying a rewrite rule with negative polarity to a hypothesis.

`Submethod:` *elim*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $member(Hyp, Hyp\_list)$

2. $rewrite\_rule(Lhs \Rightarrow Rhs)$

3. $polarity(Lhs, -)$

4. $subterm(SubHyp, Hyp)$

5. $applicable\_rewrite(Lhs \Rightarrow Rhs, SubHyp, NewSubexps)$

Postconditions:

1. $term\_update(Hyp, SubHyp, NewSubexps, NewHyps)$
2. $sequent\_update(NewHyps, Hyp\_list \vdash Goal, NewSequents)$

New goals: $NewSequents$

The preconditions of the method find a hypothesis and a rewrite rule. The rule must be applicable to a subexpression of the hypothesis.

The postconditions compute the new hypotheses; each one of them will go in a new sequent. If we assume that the number of elements in NewSubexps is n, then the predicate *term_update* produces n copies of the original hypothesis Hyp each one with the subterm $SubHyp$ replaced by a member of $NewSubexps$. The predicate *sequent_update* computes the new sequents by adding each one of the new hypotheses $NewHyps$ to the input sequent.

8.4.6 Reduce

Reduce is called by Difference Reduction. This method receives an annotated sequent and attempts to reduce the difference specified by the annotations.

`Submethod:` *reduce*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $repeat([reduce\_hyp, reduce\_goal], Subgoals)$

Postconditions: $None$

New goals: $Subgoals$

Submethod `reduce` uses one of Clam's method language predicates: *repeat*. This predicate is used to build a subplan for a given conjecture applying a fix set of methods and submethods (see [vanHarmelen *et al* 93] for details). The submethods used here

are reduce-hyp and reduce-goal. The first one ripples all annotated hypotheses and the second ripples the goal. The idea here is to try to ripple the hypotheses first and then the goal.

The reason for this is that rippling in the hypothesis list preserves the original hypotheses while new ones are generated; on the goal however, rippling literally rewrites the original goal. Since we are interested in preserving all the original skeletons marked by the annotations as they were generated in `d-reduction`, we ripple first the hypotheses because it does not risk losing any skeleton. Rippling the hypotheses often yields several subgoals. These are the outcome of twinrules and, as we saw in Chapter 7, this type of wave rule separates skeletons in different branches. Every time a wave rule is applied, the submethod `lf-ripple-hyp` (the submethod iterated by `reduce-hyp`, see below) removes from every new sequent all the annotation corresponding to the colours dropped in that branch. This way, `lf-ripple-goal` (the submethod iterated by `reduce-goal` is constrained to ripple in every branch only the colours that will lead to connections.

8.4.7 Reduce-hyp

`Reduce-hyp` is a submethods to apply recursively another submethod called `lf-ripple-hyp` which ripples an annotated hypothesis. `Reduce-hyp` ripples all possible hypotheses exhaustively.

`Submethod:` *reduce_hyp*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $repeat([lf\_ripple\_hyp], Subgoals)$

Postconditions:*none*

New goals: $Subgoals$

8.4.8 Reduce Goal

This submethod works just like `reduce-hyp`.

`Submethod:` *reduce_goal*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $repeat([lf\_ripple\_goal], Subgoals)$

Postconditions:*none*

New goals: *Subgoals*

8.4.9 lf-ripple-hyp

This submethod ripples an annotated hypothesis with a wave rule. The hypotheses are checked in the order they appear in the hypothesis list. Wave rules are also tried in the order in which they are kept in the data base.

`Submethod:` *lf_ripple_hyp*

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $member(Hyp, Hyp\_list)$
2. $annotated(Hyp)$
3. $wave\_rule(Wave\_rule)$
4. $weak\_non\_branching(Wave\_rule, Lhs \Longrightarrow Rhs)$
5. $subterm(Subexp, Hyp)$
6. $colours(Subexp, Col)$
7. $partition(Col, Col\_sub, Col\_complement)$

8. $weak(Col\_sub, Subexp, Subexp\_weak)$

9. $applicable\_wave(Lhs \Rightarrow Rhs, Subexp\_weak, NewSubexp)$

Postconditions:

1. $term\_update(Hyp, Subexp, NewSubexp, NewHyps)$

2. $NewSeqs = \{\ Hyp\_list, WH \vdash Goal \mid H \in NewHyps \wedge$
   $weak(Col\_complement, H, WH)\ \}$

   2. $Subgoals = \{\ Sub \mid S \in NewSeqs\ \wedge\ balanced\_colours(S, Sub)\ \}$

New goals:$Subgoals$

Let's first look at the preconditions. First, an annotated hypothesis Hyp is picked from the hypothesis list. Then, each subterm $Subexp$ of Hyp is tried in turn with the wave rules.

The predicate $weak\_non\_branching$ weakens the wave rule selected. It starts with the full set of colours (no weakening) and then, upon backtracking and if the rule is non-branching (i.e. not a twin-rule), it weakens the rule to non-empty subsets of colours.

Some wave rules ripple one colour and some ripple more. A wave rule is allowed to ripple a subset of the colours of a term, it does not need to ripple all of them at once. For this reason, the annotations of $Subexp$ are weakened by dropping colours from it to try to match the wave rule. Weakening the annotation of a term, amounts to removing all annotation corresponding to a set of colours. In the preconditions of the method, this is done by computing a subset of the colours of Hyp (the first subset computed is $Subexp$ itself) and weaken the term before trying to match it with the left-hand side of the rule.

In the postconditions, the new subgoals are built. The variables of the right-hand side of the selected wave rule (a list of expressions) were instantiated when Lhs was matched with $Subexp\_weak$. The first postcondition builds a list of the new hypotheses generated by the application of the wave rule. Each member of the list is a copy of Hyp with $Subexp$ replaced by a member of Rhs.

Once the new hypotheses have been put together, the new sequents are assembled. These consist of copies of the original input sequent with a new hypothesis added from $NewHyps$. The new hypothesis added to each sequent are weakened to the complement set of colours to those used in the rewriting. This complementary set of colours is obtained from the *partition* clause in the preconditions.

Finally, the set of new goals is computed. It consists of the set $NewSeqs$ with all its elements colour-balanced. This operation removes from a sequent all colours in the goal which are no longer present in the hypotheses and vice versa. The assumption is that the colours removed from a subgoal are present in another sibling subgoal and thus, in every sequent's goal, only those colours which are likely to produce a connection will be rippled.

8.4.10 lf-ripple-goal

Submethod: lf_ripple_Goal

Input: $Hyp\_list \vdash Goal$

Preconditions:

1. $annotated(Goal)$
2. $wave\_rule(Lhs \Longrightarrow Rhs)$
3. $subterm(Subexp, Goal\_weak)$
4. $colours(Subexp, Col)$
5. $partition(Col, Col\_sub, Col\_complement)$
6. $applicable\_wave(Lhs \Longrightarrow Rhs, Subexp, NewSubexp)$

Postconditions:

1. $term\_update(Goal, Subexp, NewSubexp, NewGoals)$
2. $NewSeqs = \{\ Hyp\_list \vdash WG \mid G \in NewGoals \wedge$
   $weak(Col\_complement, G, WG)\ \}$

2. $Subgoals = \{\ Sub \mid S \in NewSeqs\ \wedge\ balanced\_colours(S, Sub)\ \}$

New goals:$Subgoals$

This method is similar to the previous one, except that the polarised rippling is done on the goal. In this submethod there is no mechanism to drop colours because skeletons dropped from the goal cannot be recovered at a latter stage.

8.5 Summary

In this chapter we have described a design of a system, in the form of methods and submethods for a proof planner that specifies the techniques outlined in Chapter 4 and developed more in depth in Chapters 5, 6 and 7.

The design assumes there is a data base of wave and rewrite rules available to the methods and submethods. The methods and submethods are specifications of tactics implemented in a version of LF in the Mollusc proof editor (Section 2.2).

In the following chapter we analyse the behaviour of the specification from this chapter with some example theories and theorems and in Section 9.7 we describe an implementation of an earlier version of the methods.

Chapter 9

Examples

This chapter provides an analysis of the behaviour of the system specified in the last chapter and points to some extensions. We also mention the results obtained from the part of the system which has been implemented.

We analyse a set of logics built incrementally. We start with minimal logic as the logic containing the smallest number of inference rules and then advance by adding the appropriate rules to reach more elaborate ones. By using incrementally-built logics, we can analyse the search problems in a stepwise fashion and thus concentrate on the problems particular to each logic rather than on those inherited from the simpler ones.

There is a section for each logic. In each one of these, we first reproduce the inference rules as they are given in text books; then we give the version of these rules as they are internalised in LF. After this, a list of the rewrite rules obtained from the LF signature is given and in the following subsection the rules obtained from these are listed. This way, it is easy for the reader to refer to the various parts of the logic and for us to comment on important issues of the translations.

For every logic, the subsections after all the rules have been described present a description of how the system controls search in that logic, some example theorems and comments on the systems performance on that logic.

In Section 9.7 an implementation of an early version of the system is described and in Section 9.8, the overall performance of the system is analysed.

9.1 Minimal Propositional Logic

Minimal propositional logic is the most elementary logic we find amongst standard systems with the usual connectives. The Natural Deduction (N.D.) presentation consists of introduction and elimination rules for the main propositional connectives but does not contain rules for absurdity. Absurdity rules present particular problems that shall be discussed in Section 9.2 below.

9.1.1 Inference Rules

The following are the inference rules for minimal logic in Natural Deduction:

$$\frac{A \quad B}{A \wedge B}\ (\wedge_i) \qquad \frac{A \wedge B}{A}\ (\wedge_{el}) \qquad \frac{A \wedge B}{B}\ (\wedge_{er})$$

$$\frac{A}{A \vee B}\ (\vee_{il}) \qquad \frac{B}{A \vee B}\ (\vee_{ir}) \qquad \frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\ (\vee_e)$$

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \supset B}\ (\supset_i) \qquad \frac{A \supset B \quad A}{B}\ (\supset_e) \qquad \frac{\begin{array}{c}A\\ \vdots \\ \bot\end{array}}{\neg A}\ (\neg_i)$$

They translate into LF as follows:

$$\begin{array}{rcl}
o & : & Type \\
true & : & o \rightarrow Type \\
\bot & : & o \\
\wedge, \vee, \supset & : & o \rightarrow o \rightarrow o
\end{array}$$

$$\begin{array}{rcl}
\wedge_i & : & \Pi_{A,B:o} true(A) \rightarrow true(B) \rightarrow true(A \wedge B) \\
\wedge_{el} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(A) \\
\wedge_{er} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(B) \\
\vee_{il} & : & \Pi_{A,B:o} true(A) \rightarrow true(A \vee B) \\
\vee_{ir} & : & \Pi_{A,B:o} true(B) \rightarrow true(A \vee B) \\
\vee_e & : & \Pi_{A,B:o} true(A \vee B) \rightarrow \Pi_{C:o}((true(A) \rightarrow true(C)) \\
 & & \rightarrow (true(B) \rightarrow true(C)) \rightarrow true(C)) \\
\supset_i & : & \Pi_{A,B:o}(true(A) \rightarrow true(B)) \rightarrow true(A \supset B) \\
\supset_e & : & \Pi_{A,B:o} true(A \supset B) \rightarrow true(A) \rightarrow true(B) \\
\neg_i & : & \Pi_{A:o}(true(A) \rightarrow true(\bot)) \rightarrow true(\neg A) \\
\neg_e & : & \Pi_{A:o} true(\neg A) \rightarrow true(A) \rightarrow true(\bot)
\end{array}$$

9.1.2 Rewrite Rules

$$true(A^- \supset B^+)^+ \Longrightarrow (true(A)^- \rightarrow true(B)^+)^+ \qquad (rw{-}\supset_i)$$

$$true(A^+ \supset B^-)^- \Longrightarrow (true(A)^+ \rightarrow true(B)^-)^- \qquad (rw{-}\supset_e)$$

$$true(A^+ \wedge B^+)^+ \Longrightarrow \{true(A)^+, true(B)^+\} \qquad (rw{-}\wedge_i)$$

$$true(A^- \wedge B^-)^- \Longrightarrow true(A)^- \qquad (rw{-}\wedge_{el})$$

$$true(A^- \wedge B^-)^- \Longrightarrow true(B)^- \qquad (rw{-}\wedge_{er})$$

$$true(A^+ \vee B^+)^+ \Longrightarrow true(A)^+ \qquad (rw{-}\vee_{il})$$

$$true(A^+ \vee B^+)^+ \Longrightarrow true(B)^+ \qquad (rw{-}\vee_{ir})$$

$$true(A^- \vee B^-)^- \Longrightarrow \{true(A)^-, true(B)^-\} \qquad (rw{-}\vee_e)$$

$$true(\neg(A^-))^+ \Longrightarrow (true(A)^- \rightarrow true(\bot)^+)^+ \qquad (rw{-}\neg_i)$$

$$true(\neg(A^+))^- \Longrightarrow (true(A)^- \rightarrow true(\bot)^+)^+ \qquad (rw{-}\neg_e)$$

9.1.3 Wave Rules

$$true(\boxed{\underline{A^-}_{C_1} \supset \underline{B^+}_{C_2}}_{C_3})^+ \Longrightarrow \boxed{\underline{true(A)^-}_{C_1} \rightarrow \underline{true(B)^+}_{C_2}}_{C_3} \qquad (wr{-}\supset_i)$$

$$true(\boxed{\underline{A^+}_{C_1} \supset \underline{B^-}_{C_2}}_{C_3})^- \Longrightarrow \boxed{\underline{true(A)^+}_{C_1} \rightarrow \underline{true(B)^-}_{C_2}}_{C_3} \qquad (wr{-}\supset_e)$$

$$true(\boxed{\underline{A^+}_{C_1} \wedge \underline{B^+}_{C_2}}_{C_3})^+ \Longrightarrow \left\{ \begin{array}{l} \underline{true(A)^+}_{C_1} \\ \underline{true(B)^+}_{C_2} \end{array} \right. \qquad (wr{-}\wedge_i)$$

$$true(\boxed{\underline{A^-}_{C_1} \wedge B}_{C_1})^- \Longrightarrow \underline{true(A)^-}_{C_1} \qquad (wr{-}\wedge_{el})$$

$$true(\boxed{A \wedge \underline{B^-}_{C_2}}_{C_2})^- \Longrightarrow \underline{true(B)^-}_{C_2} \qquad (wr{-}\wedge_{er})$$

$$true(\boxed{\underline{A^+}_{C_1} \vee B}_{C_1})^+ \Longrightarrow \underline{true(A)^+}_{C_1} \qquad (wr{-}\vee_{il})$$

$$true(\boxed{A \vee \underline{B^+}_{C_2}}_{C_2})^+ \Longrightarrow \underline{true(B)^+}_{C_2} \qquad (wr{-}\vee_{ir})$$

$$true(\boxed{\underline{A^-}_{C_1} \vee \underline{B^-}_{C_2}}_{C_3})^- \Longrightarrow \left\{ \begin{array}{l} \underline{true(A)^-}_{C_1} \\ \underline{true(B)^-}_{C_2} \end{array} \right. \qquad (wr{-}\vee_e)$$

$$true(\boxed{\neg(\underline{A^-}_{C_1})}_{C_1})^+ \Longrightarrow \boxed{\underline{true(A)^-}_{C_1} \rightarrow true(\bot)^+}_{C_1} \qquad (wr{-}\neg_i)$$

$$true(\boxed{\neg(\underline{A^+}_{C_1})}_{C_1})^- \Longrightarrow \boxed{\underline{true(A)^+}_{C_1} \rightarrow true(\bot)^-}_{C_1} \qquad (wr{-}\neg_e)$$

9.1.4 Search in Minimal Logic

In minimal logic, all inference rules are either introduction or elimination rules. They all translate into rewrite rules and these into wave rules. In this sense, this logic is the one that accommodates best to our system.

In a N.D. style systems, elimination rules can be applied backwards to any term, the head of the rules is of the form $true(A)$. A similar situation occurs when introduction rules are applied forwards. Applying inference rules like this introduces a big amount of search into the proof process.

In our system this kind of search is avoided initially by converting inference rules into rewrite rules. Rewrites correspond to the application of introduction rules right-to-left and elimination rules left-to-right. This way, the connectives appear in the left-hand-side of the rules and unification constrains their applicability.

The next search point is when many rewrite rules apply to the same sequent. The system may have to choose between many rules applicable to different expressions either in the hypothesis or in the goal. Annotations constrain this possibility in two ways. First, wave rules are only applicable to annotated terms. Since **d-reduction** method annotates only one hypothesis, no rule would be applicable to any hypothesis other than the one annotated. This reduces dramatically the scope for the rewrites.

Secondly, within one expression, annotations also constrain the applicability of rules. The annotations mark the skeleton that needs to be isolated so, only wave rules that move annotations around that skeleton would be applicable. One example of this is when the goal is a conjunction (e.g. $true(a \wedge b)$). In this case, rewrite rules **rw-**$\wedge_{el}$ and **rw-**$\wedge_{er}$ are both applicable to it. If the goal is annotated, however (e.g. $true(\boxed{a \wedge \underline{b}_{\{c_1\}}})$), the corresponding wave rules (i.e. **wr-**$\wedge_{el}$ and **wr-**$\wedge_{er}$) would not both be applicable.

As we said, in minimal logic, all inference rules translate into rewrite rules and all of these parse into wave rules. In the following examples, we will observe that in all the proof plans found by the system, the method **unblock** won't be used.

9.1.5 Examples

In this section we will see some step by step examples of how our system reasons to build proof plans. We will make comments along the way to refer to the potential search problems described in the previous section.

Example 1 $true(((a \vee c) \wedge (b \vee c)) \supset ((a \wedge b) \vee c))$

Balance method gives us the following sequent:

$$\begin{aligned} hyp_1 \quad &: \quad true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ &\vdash \quad true((a^+ \wedge b^+)^+ \vee c^+)^+ \end{aligned}$$

d-unification give us the following annotations:

$$\begin{aligned} hyp_1 \quad &: \quad true(\boxed{\boxed{(\underline{a^-}_{\{c_1\}} \vee \underline{c^-}_{\{c_2\}})^-}_{\{c_1,c_2\}} \wedge \boxed{(\underline{b^-}_{\{c_3\}} \vee \underline{c^-}_{\{c_4\}})^-}_{\{c_3,c_4\}}}_{\{c_1,c_2,c_3,c_4\}})^- \\ \vdash \quad &true(\boxed{\boxed{(\underline{a^+}_{\{c_1\}} \wedge \underline{b^+}_{\{c_3\}})^+}_{\{c_1,c_3\}} \vee \underline{c^+}_{\{c_2,c_4\}}}_{\{c_1,c_2,c_3,c_4\}})^+ \end{aligned} \tag{9.1}$$

With this annotation, wave rules can ripple one colour at the time. The next step here is to ripple hyp_1 with rules **wr-**$\wedge_{el}$ and **wr-**$\wedge_{er}$:

$$\begin{aligned} hyp_1 \quad &: \quad true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 \quad &: \quad true(\boxed{\underline{a^-}_{\{c_1\}} \vee \underline{c^-}_{\{c_2\}}}_{\{c_1,c_2\}})^- \\ hyp_3 \quad &: \quad true(\boxed{(\underline{b^-}_{\{c_3\}} \vee \underline{c^-}_{\{c_4\}})}_{\{c_3,c_4\}})^- \\ \vdash \quad &true(\boxed{\boxed{(\underline{a^+}_{\{c_1\}} \wedge \underline{b^+}_{\{c_3\}})^+}_{\{c_1,c_3\}} \vee \underline{c^+}_{\{c_2,c_4\}}}_{\{c_1,c_2,c_3,c_4\}})^+ \end{aligned}$$

Rippling hyp_2 with twin rule **wr-**$\vee_e$ produces 2 sequents:

$$
\begin{array}{lcl}
hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\
hyp_2 & : & true(a^- \vee c^-)^- \\
hyp_3 & : & true(\underline{\boxed{(\underline{b^-}_{\{c_3\}} \vee \underline{c^-}_{\{c_4\}})}}_{\{c_3,c_4\}})^- \\
hyp_4 & : & \underline{true(a)^-}_{\{c_1\}}
\end{array}
$$

$$
\vdash \; true(\underline{\underline{\boxed{(\underline{a^+}_{\{c_1\}} \wedge \underline{b^+}_{\{c_3\}})^+}}_{\{c_1,c_3\}} \vee \underline{c^+}_{\{c_4\}}}_{\{c_1,c_3,c_4\}})^+
$$

$$
\begin{array}{lcl}
hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\
hyp_2 & : & true(a^- \vee c^-)^- \\
hyp_3 & : & true(\underline{\boxed{(\underline{b^-}_{\{c_3\}} \vee \underline{c^-}_{\{c_4\}})}}_{\{c_3,c_4\}})^- \\
hyp_4 & : & \underline{true(c)^-}_{\{c_2\}}
\end{array}
$$

$$
\vdash \; true(\underline{\underline{\boxed{(a^+ \wedge \underline{b^+}_{\{c_3\}})^+}}_{\{c_3\}} \vee \underline{c^+}_{\{c_2,c_4\}}}_{\{c_2,c_3,c_4\}})^+
$$

Notice that in each one of these two previous sequents the colours dropped by the rule application are also dropped in the goal. This prevents the system from being misled in rippling towards a skeleton that won't make a connection any more because it has been dropped from the hypothesis list.

Rippling hyp_3 in both sequents above we obtain 4 new sequents:

$$
\begin{array}{lcl}
hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\
hyp_2 & : & true(a^- \vee c^-)^- \\
hyp_3 & : & true((b^- \vee c^-))^- \\
hyp_4 & : & \underline{true(a)^-}_{\{c_1\}} \\
hyp_5 & : & \underline{true(b)^-}_{\{c_3\}}
\end{array}
\tag{9.2}
$$

$$
\vdash \; true(\underline{\underline{\boxed{(\underline{a^+}_{\{c_1\}} \wedge \underline{b^+}_{\{c_3\}})^+}}_{\{c_1,c_3\}} \vee c^+}_{\{c_1,c_3\}})^+
$$

$$
\begin{array}{lcl}
hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\
hyp_2 & : & true(a^- \vee c^-)^- \\
hyp_3 & : & true((b^- \vee c^-))^- \\
hyp_4 & : & \underline{true(a)^-}_{\{c_1\}} \\
hyp_5 & : & \underline{true(c)^-}_{\{c_4\}}
\end{array}
\tag{9.3}
$$

$$
\vdash \; true(\underline{\underline{\boxed{(\underline{a^+}_{\{c_1\}} \wedge b^+)^+}}_{\{c_1\}} \vee \underline{c^+}_{\{c_4\}}}_{\{c_1,c_4\}})^+
$$

$$\begin{array}{lcl} hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 & : & true(a^- \vee c^-)^- \\ hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(c)^-}_{\{c_2\}} \\ hyp_5 & : & \underline{true(b)^-}_{\{c_3\}} \end{array} \tag{9.4}$$

$$\vdash \; true(\underline{\boxed{\boxed{(a^+ \wedge \underline{b^+}_{\{c_3\}})^+}_{\{c_3\}} \vee \underline{c^+}_{\{c_2\}}}}_{\{c_2,c_3\}})^+$$

$$\begin{array}{lcl} hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 & : & true(a^- \vee c^-)^- \\ hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(c)^-}_{\{c_2\}} \\ hyp_5 & : & \underline{true(c)^-}_{\{c_4\}} \end{array} \tag{9.5}$$

$$\vdash \; true(\underline{\boxed{(a^+ \wedge b^+)^+ \vee \underline{c^+}_{\{c_2,c_4\}}}}_{\{c_2,c_4\}})^+$$

At this point, all skeletons have been rippled in the hypothesis list. Now rippling starts in the goals of all the sequents. In Sequent 9.2, the goal is rippled with **wr-**$\vee_{il}$:

$$\begin{array}{lcl} hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(a)^-}_{\{c_1\}} \\ hyp_5 & : & \underline{true(b)^-}_{\{c_3\}} \end{array}$$

$$\vdash \; true(\underline{\boxed{\underline{a^+}_{\{c_1\}} \wedge \underline{b^+}_{\{c_3\}}}}_{\{c_1,c_3\}})^+$$

and then with rule **wr-**$\wedge_i$ leaving two axioms:

$$\begin{array}{lcl} hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(a)^-}_{\{c_1\}} \\ hyp_5 & : & true(b)^- \end{array}$$

$$\vdash \; \underline{true(a^+)^+}_{\{c_1\}}$$

$$\begin{array}{lcl} hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & true(a)^- \\ hyp_5 & : & \underline{true(b)^-}_{\{c_3\}} \end{array}$$

$$\vdash \; \underline{true(b^+)^+}_{\{c_3\}}$$

In Sequent 9.5, the other disjunct is rippled and an axiom is obtained and method **axiom** applies:

$$\begin{array}{lcl} hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 & : & true(a^- \vee c^-)^- \\ hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(c)^-}_{\{c_2\}} \\ hyp_5 & : & \underline{true(c)^-}_{\{c_4\}} \\ & & \\ & \vdash & \underline{true(c^+)^+}_{\{c_2,c_4\}} \end{array}$$

Sequents 9.3 and 9.4 cannot be rippled because there is one colour in every disjunct. Rules **wr-**$\vee_{il}$ and **wr-**$\vee_{ir}$ do not apply because the colour on the disjunct not being rippled would have to be dropped and this is not done by submethod **lf_ripple_goal**. This finishes **d_reduction**.

The planner tries **balance** again but it does not succeed. The next method applicable to Sequents 9.2 and 9.3 is again **d_reduction**. The annotations obtained in them are the following:

$$\begin{array}{lcl} hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 & : & true(a^- \vee c^-)^- \\ hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(a)^-}_{\{c_1\}} \\ hyp_5 & : & true(c)^- \\ & & \\ & \vdash & true(\underline{\boxed{\underline{(a^+_{\{c_1\}} \wedge b^+)^+}_{\{c_1\}} \vee c^+}}_{\{c_1\}})^+ \end{array} \tag{9.6}$$

$$\begin{array}{lcl} hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 & : & true(a^- \vee c^-)^- \\ hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(c)^-}_{\{c_1\}} \\ hyp_5 & : & true(b)^- \\ & & \\ & \vdash & true(\underline{\boxed{(a^+ \wedge b^+)^+ \vee \underline{c^+}_{\{c_1\}}}}_{\{c_1\}})^+ \end{array} \tag{9.7}$$

These annotations are obtained because all the other hypotheses that could be used have already been rippled. In the Sequent 9.7, the goal is rippled with rule **wr-**$\vee_{ir}$ and an axiom is obtained. In Sequent 9.7, although rule **wr-**$\vee_{il}$ applies, the skeleton

cannot be completely rippled because rule **wr-**$\wedge_i$ does not apply:

$$\begin{array}{lcl} hyp_1 & : & true((a^- \vee c^-)^- \wedge (b^- \vee c^-)^-)^- \\ hyp_2 & : & true(a^- \vee c^-)^- \\ hyp_3 & : & true((b^- \vee c^-))^- \\ hyp_4 & : & \underline{true(a)^-}_{\{c_1\}} \\ hyp_5 & : & true(c)^- \end{array}$$

$$\vdash \; true(\boxed{\boxed{(\underline{a^+}_{\{c_1\}} \wedge b^+)^+}}_{\{c_1\}})^+$$

This forces the planner to backtrack and take annother annotation in Sequent 9.6. This time the goal will match hypothesis hyp_5 and the skeleton will be $true(c)$. With this annotation, as in Sequent 9.7, the goal can be ripple to obtain an axiom.

Example 2 $true(a \supset \neg\neg a)$

Our third example involves negation. It is a simple linear example to show how the definition of negation (i.e. $true(\neg a) \equiv true(a) \rightarrow true(\bot)$) is *unfolded* by rippling. The planner starts the process, as usual, by balancing. **D_reduction** annotates the sequent as follows:

$$\begin{array}{lcl} hyp_1 & : & true(a)^- \end{array}$$

$$\vdash \; true(\boxed{\neg\boxed{(\neg\underline{a^+}_{\{c_1\}})^-}_{\{c_1\}}}_{\{c_1\}})^+$$

Now, no rippling is possible in the hypotheses so the next submethod is **lf_ripple_goal**. Rule **wr-**$\neg_i$ applies to the goal:

$$\begin{array}{lcl} hyp_1 & : & true(a)^- \end{array}$$

$$\vdash \; \boxed{true(\boxed{\neg\underline{a^+}_{\{c_1\}}}_{\{c_1\}})^- \rightarrow true(\bot)^+}_{\{c_1\}}$$

and then rule **wr-**$\neg_e$ to the inner wave front:

$$\begin{array}{lcl} hyp_1 & : & true(a)^- \end{array}$$

$$\vdash \; \boxed{\boxed{\underline{true(a)^+}_{\{c_1\}} \rightarrow true(\bot)^-)^-}_{\{c_1\}} \rightarrow true(\bot)^+}_{\{c_1\}}$$

There is no more rippling possible so planning continues with the next applicable method which is **refine_ctxt**. Weak-fertilising the head of the goal with the antecedent of the goal gives:

$$\begin{array}{lcl} hyp_1 & : & true(a)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(\bot)^- \\ \\ & \vdash & true(a)^+ \end{array}$$

which is an axiom.

Example 3 $true((a \supset c) \wedge (b \supset c) \wedge (a \vee b) \supset c)$

In this example, *cautious* balance leads the planner through the wrong path. What is needed is *eager* balance. We first describe the behaviour of the planner with the first kind of balance and then with the second after backtracking.

After balancing and starting difference reduction, the sequent looks like this:

$$hyp_1 \quad : \quad true(\boxed{\boxed{(a^+ \supset \underline{c^-}_{\{c_1\}})^-}_{\{c_1\}} \wedge \boxed{(b^+ \supset \underline{c^-}_{\{c_2\}})^-}_{\{c_2\}}}_{\{c_1,c_2\}} \wedge (a^- \vee b^-)^- \quad)^-_{\{c_1,c_2\}}$$

$$\vdash \underline{true(c)^+}_{\{c_1,c_2\}}$$

Rippling hyp_1 with rules **wr-**$\wedge_{er}$, **wr-**$\wedge_{el}$ and **wr-**$\supset_e$:

$$\begin{array}{lcl} hyp_1 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^- \wedge (a^- \vee b^-)^-)^- \\ hyp_2 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^-)^- \\ hyp_3 & : & true(a^+ \supset c^-)^- \\ hyp_4 & : & true(b^+ \supset c^-)^- \\ hyp_5 & : & \boxed{true(a)^+ \rightarrow \underline{true(c)^-}_{\{c_1\}}}_{\{c_1\}} \\ hyp_6 & : & \boxed{true(b)^+ \rightarrow \underline{true(c)^-}_{\{c_2\}}}_{\{c_2\}} \\ \\ & \vdash & \underline{true(c)^+}_{\{c_1,c_2\}} \end{array}$$

and refining with hyp_6 we obtain $true(b)$ as a new goal:

$$\begin{array}{lcl} hyp_1 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^- \wedge (a^- \vee b^-)^-)^- \\ hyp_2 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^-)^- \\ hyp_3 & : & true(a^+ \supset c^-)^- \\ hyp_4 & : & true(a^+ \supset c^-)^- \\ hyp_5 & : & true(a)^+ \rightarrow true(c)^- \\ hyp_6 & : & true(b)^+ \rightarrow true(c)^- \\ \\ & \vdash & true(b)^+ \end{array}$$

At this stage difference reduction will start again identifying the only possible connection:

$$
\begin{array}{lcl}
hyp_1 & : & true(\boxed{(a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^- \wedge \underline{\boxed{(a^- \vee \underline{b^-}_{\{c_1\}})^-}}_{\{c_1\}}}_{\{c_1\}}) \\
hyp_2 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_3 & : & true(a^+ \supset c^-)^- \\
hyp_4 & : & true(a^+ \supset c^-)^- \\
hyp_5 & : & true(a)^+ \rightarrow true(c)^- \\
hyp_6 & : & true(b)^+ \rightarrow true(c)^- \\
 & \vdash & true(b)^+
\end{array}
$$

Now **d-reduction** will ripple hyp_1 with rule **wr-**$\wedge_{er}$:

$$
\begin{array}{lcl}
hyp_5 & : & true(a)^+ \rightarrow true(c)^- \\
hyp_6 & : & true(b)^+ \rightarrow true(c)^- \\
hyp_7 & : & true(\underline{\boxed{a^- \vee \underline{b^-}_{\{c_1\}}}}_{\{c_1\}})^- \\
 & \vdash & true(b)^+
\end{array}
$$

Since hyp_7 cannot be rippled because no wave rule is applicable, in particular **wr-**$\vee_e$. This finishes the reduction step of **d-reduction** and the planning process continues.

No method is applicable to this sequent and the planner is forced to backtrack until the second clause of **balance** is tried. *Eager* **balance** produces the following sequents:

$$
\begin{array}{lcl}
hyp_1 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^- \wedge (a^- \vee b^-)^-)^- \\
hyp_2 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_3 & : & true(a^+ \supset c^-)^- \\
hyp_4 & : & true(a)^+ \rightarrow true(c)^- \\
hyp_5 & : & true(b^+ \supset c^-)^- \\
hyp_6 & : & true(b)^- \rightarrow true(c)^- \\
hyp_7 & : & true(a^- \vee b^-)^- \\
hyp_8 & : & true(a)^- \\
 & \vdash & true(c)^+
\end{array}
$$

$$
\begin{array}{lcl}
hyp_1 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^- \wedge (a^- \vee b^-)^-)^- \\
hyp_2 & : & true((a^+ \supset c^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_3 & : & true(a^+ \supset c^-)^- \\
hyp_4 & : & true(a)^+ \rightarrow true(c)^- \\
hyp_5 & : & true(b^+ \supset c^-)^- \\
hyp_6 & : & true(b)^+ \rightarrow true(c)^- \\
hyp_7 & : & true(a^- \vee b^-)^- \\
hyp_8 & : & true(b)^- \\
 & & \\
 & \vdash & true(c)^+
\end{array}
$$

and continue planning by weak-fertilising hyp_4 on hyp_8 and hyp_6 on hyp_8 respectively to obtain:

$$
\begin{array}{lcl}
hyp_7 & : & true(a^- \vee b^-)^- \\
hyp_8 & : & true(a)^- \\
hyp_9 & : & true(c)^- \\
 & & \\
 & \vdash & true(c)^+
\end{array}
$$

$$
\begin{array}{lcl}
hyp_7 & : & true(a^- \vee b^-)^- \\
hyp_8 & : & true(b)^- \\
hyp_9 & : & true(c)^- \\
 & & \\
 & \vdash & true(c)^+
\end{array}
$$

These two sequents are axioms. This theorem is proved without rippling; only **balance** and **weak-fertilise** are needed.

9.1.6 Discussion

The examples in the previous section show that the ideas behind our system work in practice.

The example we saw in Section 4.3 is linear: every rule application generates one subgoal only. First, connections a and c are identified with colours c_1 and c_2. Then the two colours are rippled in a single branch going from top to bottom in a *breadth first* way. Hyp_1 is rippled in two stages (hyp_2 and hyp_3) and only when these have been finished, are the new hypotheses rippled. After these two colours are fully rippled, weak-fertilisation make the connection on c then on b and finally on a.

Example 1 allows us to see how coloured annotated terms keep track of connections in a branching proof. Rippling in the hypothesis isolates the connections in different branches and colours guide the application of wave rules to obtain the neccessary connections from the goal.

At each stage of the proof, rewrite rules are applicable to all the hypotheses. Without annotations, the search space is enormous; every disjunction in the hypotheses and every conjunction in the goal breaks the proof in two and many of the combinations don't lead to closing the branches. The annotations control the applicability of the rules by keeping track of the connections and preventing the application of rules that lead to unprovable sequents.

In Example 3, the planner backtracks to apply eager balance and prove the theorem. After applying cautious balance, annotations identify the two conjuncts where the connections with c are but do not identify the connections with a and b because these get introduced into the hypothesis list. Annotations guide the isolation of the two connections in c but the choice of one or the other prevents relisation of both connections. We still don't have a general strategy for the **balance** method. In most of the theorems we have experimented on, *cautious* balance is the best strategy to follow but there are theorems where **balance** is unable to distribute the connecting formulae across the sequent symbol. Eager balance and step balance complement cautious balance.

Minimal propositional logic is the core of the logics used in the following sections. The aspects of proof search in this logic we have just discussed, apply to the following logics too.

9.2 Intuitionistic Propositional Logic

In intuitionistic propositional logic absurdity starts playing a special role. $\bot$ is not just any other constant of the language, it is a distinguished constant because it has special properties within the system. The property of absurdity ($\bot$) relevant to intuitionistic logic will be represented in the inference rule introduced in the next section.

9.2.1 Inference Rules

We can obtain intuitionistic propositional logic from minimal propositional logic by adding to those described in section 9.1.1 the following inference rule:

$$\frac{\bot}{a}$$

which is encoded in LF as:

$$abs : \Pi_{a:o} true(\bot) \rightarrow true(a)$$

This rule, called rule of *intuitionistic absurdity* is usually read as "from a contradiction (or absurdity) anything is derivable" but, if we want to use it backwards we must read it as "one way of proving a formula is by showing that the assumptions are inconsistent (or absurd)".

9.2.2 Rewrite Rules

The rule described above can be used in lf, as described by the reading of it above, forwards or backwards. However, when we want to use the rules in proof search, rewriting absurdity into *anything* is not of much help. A specific formula must be specified and this is hard to do in advance. For the backwards case, it seems plausible to rewrite *a* formula into absurdity as a way of allowing the system to take the *path* of proving a formula by trying to show the assumptions are inconsistent. The acceptable rewrite rule is therefore:

$$true(a)^+ \Longrightarrow true(\bot)^+ \qquad (rw{-}\bot)$$

9.2.3 Search in Propositional Intuitionistic Logic

Rule **rw-⊥** is hard to control. In principle, it could be applicable to any formula with positive polarity. Since its left-hand side and its right-hand side are not d-unifiable, it cannot be made into a wave rule. Therefore, this rule will be used only by the **unblock** method.

9.2.4 Examples

The selection of examples we show here are theorems which are provable in intuitionistic propositional logic but not in minimal logic.

Example 4 $true(\neg\neg(\neg\neg a \supset a))$

To start, **balance** method does one introduction step:

$$\begin{array}{rcl} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ & \vdash & true(\bot)^+ \end{array}$$

no difference unification is possible because $\bot$ does not appear in the hypotheses, so **d-reduction** fails. The next applicable method is then **unblock**. The rule this method applies is **rw-$\neg_e$** because it allows weak-fertilisation.

$$\begin{array}{rcl} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ hyp_2 & : & (true((\neg(\neg a^-)^+)^- \supset a^+)^+ \rightarrow true(\bot)^-)^- \\ & \vdash & true(\bot)^+ \end{array}$$

After **weak-fertilise** has been applied, the sequent look like this:

$$\begin{array}{rcl} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ hyp_2 & : & (true((\neg(\neg a^-)^+)^- \supset a^+)^+ \rightarrow true(\bot)^-)^- \\ & \vdash & true((\neg(\neg a^-)^+)^- \supset a^+)^+ \end{array}$$

balance applies again:

$$\begin{array}{rcl} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ hyp_2 & : & (true((\neg(\neg a^-)^+)^- \supset a^+)^+ \rightarrow true(\bot)^-)^- \\ hyp_3 & : & true((\neg(\neg a^-)^+)^-)^- \\ & \vdash & true(a)^+ \end{array}$$

D_reduction starts by d-unifying the goal with all the hypothesis. The following solutions are found:

$$true(\boxed{\neg(\boxed{(\neg\boxed{\underline{a^-}_{c_1}}^{+})^{-}}_{\{c_1\}} \supset a^+)^-}_{\{c_1\}})^- \equiv_{du} true(a)^+ \qquad (hyp_1 - goal)$$

$$\boxed{(true(\boxed{\neg\boxed{(\neg\underline{a^-}_{\{c_1\}})^+}_{\{c_1\}} \supset a^+}_{\{c_1\}})^+ \rightarrow true(\bot)^-)^-}_{\{c_1\}} \equiv_{du} true(a)^+$$

$$(hyp_2 - goal)$$

$$true(\boxed{\neg\boxed{(\neg\underline{a^-}_{c_1})^+}_{\{c_1\}}}_{\{c_1\}})^- \equiv_{du} true(a)^+ \qquad (hyp_3 - goal)$$

Amongst all the possibilities, hyp_3 has the lowest wave measure, so it is selected for rippling. **Reduce-hyp** ripples hyp_3 with rule **wr-**$\neg_e$ twice:

$$\begin{array}{lll} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ hyp_2 & : & (true((\neg(\neg a^-)^+)^- \supset a^+)^+ \rightarrow true(\bot)^-)^- \\ hyp_3 & : & true(\neg(\neg a^-)^+)^- \\ hyp_4 & : & true(\neg a^-)^+ \rightarrow true(\bot)^- \\ hyp_5 & : & \boxed{(\boxed{\underline{(true(a)^-}_{\{c_1\}} \rightarrow true(\bot)^+)^+}_{\{c_1\}} \rightarrow true(\bot)^-)^-}_{\{c_1\}} \\ \\ & \vdash & true(a)^+ \end{array}$$

No rippling is possible in the goal so **reduce-goal** fails. The next applicable method is **unblock**. **D-reduction** does not apply because all hypothesis have been rippled. **Unblock** applies rule **rw-**$\bot$ to enable fertilisation:

$$\begin{array}{lll} hyp_4 & : & true(\neg a^-)^+ \rightarrow true(\bot)^- \\ hyp_5 & : & ((true(a)^+ \rightarrow true(\bot)^-)^+ \rightarrow true(\bot)^-)^- \\ \\ & \vdash & true(\bot)^+ \end{array}$$

weak-fertilising with hyp_5 yields:

$$\begin{array}{lll} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ hyp_2 & : & (true((\neg(\neg a^-)^+)^- \supset a^+)^+ \rightarrow true(\bot)^-)^- \\ hyp_3 & : & true(\neg(\neg a^-)^+)^- \\ hyp_4 & : & true(\neg a^-)^+ \rightarrow true(\bot)^- \\ hyp_5 & : & ((true(a)^+ \rightarrow true(\bot)^-)^+ \rightarrow true(\bot)^-)^- \\ \\ & \vdash & true(a)^- \rightarrow true(\bot)^+ \end{array}$$

and again with hyp_2:

$$\begin{array}{lcl} hyp_1 & : & true(\neg((\neg(\neg a^-)^+)^- \supset a^+)^+)^- \\ hyp_2 & : & (true((\neg(\neg a^-)^+)^- \supset a^+)^+ \rightarrow true(\bot)^-)^- \\ hyp_3 & : & true(\neg(\neg a^-)^+)^- \\ hyp_4 & : & true(\neg a)^- \rightarrow true(\bot)^- \\ hyp_5 & : & ((true(a)^+ \rightarrow true(\bot)^-)^+ \rightarrow true(\bot)^-)^- \\ hyp_6 & : & true(a)^- \\ \\ & \vdash & true(\neg\neg a \supset a)^+ \end{array}$$

finally, **balance** applies and produces an axiom:

$$\begin{array}{lcl} hyp_4 & : & true(\neg a)^- \rightarrow true(\bot)^- \\ hyp_5 & : & ((true(a)^+ \rightarrow true(\bot)^-)^+ \rightarrow true(\bot)^-)^- \\ hyp_6 & : & true(a)^- \\ hyp_7 & : & true(\neg(\neg a^-)^+)^- \\ \\ & \vdash & true(a)^+ \end{array}$$

Example 5 $true((\neg\neg a \supset \neg\neg b) \supset \neg\neg(a \supset b))$

As usual, the planner first balances and starts difference reduction by difference unifying the goal and hypothesis:

$$hyp_1 \; : \; true(\boxed{(\neg\boxed{(\neg\underline{a^+}_{c_1})^-}_{\{c_1\}})^+ \supset \neg\neg\underline{b^-}_{c_2}}_{\{c_1\}})^-$$

$$\vdash \; true(\boxed{(\neg\boxed{(\neg\underline{a^-}_{c_1})^+}_{\{c_1\}})^- \supset \underline{b^+}_{c_2}}_{\{c_1\}})^+$$

Rippling both hypothesis and goal takes us a long way in this example

$$\begin{array}{lcl} hyp_1 & : & true((\neg(\neg a^+)^-)^+ \supset (\neg(\neg b^-)^+)^-)^- \\ hyp_2 & : & true(\neg(\neg a^+)^-)^+ \rightarrow true(\neg(\neg b^-)^+)^- \\ hyp_3 & : & (true(\neg a^+)^- \rightarrow true(\bot)^+)^+ \rightarrow true(\neg(\neg b^-)^+)^- \\ hyp_4 & : & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \rightarrow true(\neg(\neg b^-)^+)^- \\ hyp_5 & : & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \rightarrow \underline{(true(\neg b^-)^+ \rightarrow true(\bot)^-)^-} \\ hyp_6 & : & \boxed{((\underline{true(a)^+}_{\{c_1\}} \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \rightarrow \underline{\boxed{(\underline{true(b)^-}_{\{c_2\}} \rightarrow true(\bot)^+)^+ \rightarrow true(\bot)^-}}_{\{c_1\}}} \\ \\ & \vdash & \boxed{((\underline{true(a)^-}^-_{\{c_1\}} \rightarrow \underline{true(b)^+}_{\{c_2\}})^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+}_{\{c_1\}} \end{array}$$

in which the sequent is fully rippled. Weak fertilisation is possible using hyp_6. Two sequents are produced:

$$\begin{array}{lll} hyp_6 & : & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \rightarrow (true(b)^- \rightarrow true(\bot)^+)^+ \rightarrow true(\bot)^- \\ hyp_7 & : & ((true(a)^- \rightarrow true(b)^+)^+ \rightarrow true(\bot)^-)^- \\ \\ & \vdash & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \\ & \vdash & (true(b)^- \rightarrow true(\bot)^+)^+ \end{array}$$

Weak fertilisation applies again to both sequents, this time with hypothesis hyp_7:

$$\begin{array}{lll} hyp_6 & : & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \rightarrow (true(b)^- \rightarrow true(\bot)^+)^+ \rightarrow true(\bot)^- \\ hyp_7 & : & ((true(a)^- \rightarrow true(b)^+) \rightarrow true(\bot)^-)^- \\ & & hyp_8 : (true(a)^+ \rightarrow true(\bot)^-)^- \\ \\ & \vdash & (true(a)^- \rightarrow true(b)^+)^+ \\ \\ & & hyp_8 : true(b)^- \\ \\ & \vdash & (true(a)^- \rightarrow true(b)^+)^+ \end{array}$$

The second sequent above is trivial and method **axiom** applies. For the first sequent the only applicable method is **unblock**. The rule used to enable fertilisation is again **rw-⊥**:

$$\begin{array}{lll} hyp_6 & : & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+)^+ \rightarrow (true(b)^- \rightarrow true(\bot)^+)^+ \rightarrow true(\bot)^- \\ hyp_7 & : & ((true(a)^- \rightarrow true(b)^+)^+ \rightarrow true(\bot)^-)^- \\ & & hyp_8 : (true(a)^+ \rightarrow true(\bot)^-)^- \\ & & hyp_9 : true(a)^- \\ \\ & \vdash & true(\bot)^+ \end{array}$$

The introduction of $true(\bot)$ as a goal, similar to Example 4, enables weak-fertilisation. In this case it is with hyp_8:

$$\begin{array}{lll} hyp_6 & : & ((true(a)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^-)^- \rightarrow (true(b)^- \rightarrow true(\bot)^+)^+ \rightarrow true(\bot)^- \\ hyp_7 & : & ((true(a)^- \rightarrow true(b)^+) \rightarrow true(\bot))^- \\ hyp_8 & : & (true(a)^+ \rightarrow true(\bot)^-)^- \\ hyp_9 & : & true(a)^- \\ \\ & \vdash & true(a)^+ \end{array}$$

and this is an axiom.

9.2.5 Discussion

In Intuitionistic propositional logic we have rule (**rw-⊥**) which cannot be converted into a wave rule because there are no common skeletons on both sides of the rule.

Therefore, the rule for absurdity has to be used as a rewrite rule.

In the examples of the previous section, we saw method **unblock** in use. Rules like **rw-**$\neg_e$ and **rw-***bot* where used to enable weak fertilisation. Rule **rw-**$\bot$ is unconstrained and so it could be applied to any judgement with the same judgement function. The first clause of method **unblock** however looks for rewrite rules which enable fertilisation and, as we saw in the examples, this restriction often finds the appropriate place to apply rules like **rw-**$\bot$ with little syntactic guidance.

Rule **rw-**$\bot$ introduces a contradiction symbol in the goal and so the context has to be proven inconsistent. Applying this rule when it enables weak-fertilisation means that there is at least one inference rule in the context whose head is $true(\bot)$; that is, there is already a rule in the context which establishes some conditions for the context to be inconsistent. For this reason, the restriction in **unblock** is useful to find the right place for the rule to be applied.

Most of the rules of the logic are used, as in Minimal logic, as wave rules and in a few places where non-difference-reducing steps are needed, method **unblock** finds the appropriate rule to be applied.

9.3 Classical Propositional Logic

Classical Logic is different from intuitionistic logic in that it allows proving theorems by *reductio ad absurdum*. That is, it is possible to assume that some proposition is *not* the case, deduce a contradiction ($\bot$) and conclude that the original proposition is true.

9.3.1 Inference Rules

There are several ways of obtaining a N.D. presentation of classical propositional logic. The most common ones are the addition either of the following two inference rules to the rules of intuitionistic logic (*c.f.* Section 9.2.1).

The first one is called *Elimination of Double Negation*($\neg\neg_e$):

$$\frac{\neg\neg A}{A}$$

and the second one is called the rule of *Classical Absurdity* or *reductio ad absurdum* (raa):

$$\dfrac{\begin{array}{c}\neg A\\ \vdots\\ \bot\end{array}}{A}$$

These rules are encoded in LF as follows. $\neg\neg_e$:

$$\Pi_{a:o} true(\neg\neg a) \rightarrow true(a)$$

and raa:

$$\Pi_{a:o}(true(\neg a) \rightarrow true(\bot)) \rightarrow true(a)$$

9.3.2 Rewrite Rules

The rule $\neg\neg_e$ gives the following rewrite:

$$true(\neg(\neg A^-)^+)^- \Longrightarrow true(A)^- \qquad (rw{-}\neg\neg_e)$$

and raa:

$$true(A)^+ \Longrightarrow (true(\neg A^+)^- \rightarrow true(\bot)^+)^+ \qquad (rw{-}raa)$$

9.3.3 Wave Rules

The rule $\neg\neg_e$ gives the following wave rule:

$$true(\underline{\boxed{\neg(\neg \underline{A^-}_{C_1})^+}}_{C_1})^- \Longrightarrow \underline{true(A)^-}_{C_1} \qquad (wr{-}\neg\neg_e)$$

9.3.4 Search in Propositional Classical Logic

We have extended now our wave rule database with one more rule (**wr-**$\neg\neg_e$), though we don't use it often in practice. Rule **rw-**raa, whilst not a wave rule, can be used for unblocking purposes. One of the differences in proof techniques between intuitionistic and classical logics is that in the former we only *use* a goal once whereas in the latter, a goal may need to be used more than once. Rule **rw-**raa is the vehicle to *store* the goal in the hypothesis list for further use. For example, if we have a goal $true(a \vee b)$,

in Propositional logic we must choose one disjunct when applying rules **rw-**$\vee_{il}$ and **rw-**$\vee_{ir}$ and the disjunct which wasn't selected will be lost.

In classical logic, however, it is possible to use rule **rw-**raa to store the goal as follows. First, apply **rw-**raa:

$$\vdash (true(\neg(a^+ \vee b^+)^+)^- \rightarrow true(\bot)^+)^+$$

then, introduce $\rightarrow$:

$$true(\neg(a^+ \vee b^+)^+)^- \vdash true(\bot)^+)^+$$

At this point, the disjunction has been *stored* (negated) in the hypotheses and we can recover it by using rule **rw-**$\neg_e$ on the hypothesis and then refining:

$$\begin{array}{lcl} hyp_1 & : & true(\neg(a^+ \vee b^+)^+)^- \\ hyp_2 & : & true(a^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ & \vdash & true(a^+ \vee b^+)^+ \end{array}$$

Now it is possible to select the left disjunct (say) from the goal:

$$\begin{array}{lcl} hyp_1 & : & true(\neg(a^+ \vee b^+)^+)^- \\ hyp_2 & : & true(a^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ & \vdash & true(a)^+ \end{array}$$

and later in the proof, when the original goal is needed again, it is possible to use rule **rw-**$\bot$ to introduce $\bot$ in the goal:

$$\begin{array}{lcl} hyp_1 & : & true(\neg(a^+ \vee b^+)^+)^- \\ hyp_2 & : & true(a^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ & \vdash & true(\bot)^+ \end{array}$$

and recover the original disjunction by refinement:

$$\begin{array}{lcl} hyp_1 & : & true(\neg(a^+ \vee b^+)^+)^- \\ hyp_2 & : & true(a^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ & \vdash & true(a^+ \vee b^+)^+ \end{array}$$

This need to *reuse* goals is also present in Sequent Calculus[Gentzen 69] where the difference between intuitionistic and classical presentations consists of allowing one goal, in the former, and many in the latter.

9.3.5 Examples

In this section, as before for the intuitionistic case, we will describe a theorem which is provable in classical logic but not in intuitionistic or minimal logics.

Example 6 $true((a \supset b) \supset (\neg a \vee b))$

To start the proof, **balance** introduces the implication and **d-reduction** d-unifies goal and hypothesis:

$$\begin{array}{rcl} hyp_1 & : & true(\boxed{\underline{a^+}_{\{c_1\}} \supset \underline{b^-}_{\{c_2\}}}_{\{c_1,c_2\}})^- \\ \\ & \vdash & true(\boxed{\boxed{\neg\underline{a}_{\{c_1\}}}_{\{c_1\}} \vee \underline{b^+}_{\{c_2\}}}_{\{c_1,c_2\}})^+ \end{array}$$

Rippling the hypothesis is possible but the goal cannot be rippled because the applicable rules would drop a colour.

$$\begin{array}{rcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & \boxed{\underline{true(a)^+}_{\{c_1\}} \rightarrow \underline{true(b)^-}_{\{c_2\}}}_{\{c_1,c_2\}} \\ \\ & \vdash & true(\boxed{(\neg\underline{a^+}_{\{c_1\}})^- \vee \underline{b^+}_{\{c_2\}}}_{\{c_1,c_2\}})^+ \end{array}$$

Method **unblock** applies rewrite rule **rw-**raa:

$$\begin{array}{rcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ \\ & \vdash & true(\neg(\neg a \vee b))^- \rightarrow true(\bot) \end{array}$$

No other method is applicable so **unblock** applies again, this time with rewrite **rw-**$\neg_e$ on hyp_3:

$$\begin{array}{rcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ \\ & \vdash & (true(\neg a \vee b)^+ \rightarrow true(\bot)^-)^- \rightarrow true(\bot)^+ \end{array}$$

This time, **balance** is applicable:

$$\begin{array}{rcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ \\ & \vdash & true(\bot)^+ \end{array}$$

Weak-fertilising recovers the original disjunction and **d_reduction** re-annotates goal and hypothesis:

$$\begin{array}{lcl} hyp_1 & : & true(\boxed{\underline{a^+}_{\{c_1\}} \supset \underline{b^-}_{\{c_2\}}}_{\{c_1,c_2\}})^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ \\ & \vdash & true(\boxed{\underline{(\neg\underline{a^-}_{\{c_1\}})^+ \vee \underline{b^+}_{\{c_2\}}}}_{\{c_1,c_2\}})^+ \end{array} \tag{9.8}$$

Rippling either the hypotheses or the conclusion is not possible: hyp_1 has already been rippled and the goal has to many skeletons for either **wr-**or_{il} or **wr-**or_{ir} to apply. **D-reduction** backtracks over the list of pairs of annotated goal and hypotheses and selects a weaker annotation with only one colour:

$$\begin{array}{lcl} hyp_1 & : & true(\boxed{\underline{a^+}_{\{c_1\}} \supset b}_{\{c_1\}})^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ \\ & \vdash & true(\boxed{\underline{(\neg\underline{a^-}_{\{c_1\}})^+ \vee b^+}}_{\{c_1\}})^+ \end{array}$$

The hypotheses cannot be rippled; The goal is rippled, first with rule **wr-**$\vee_{el}$:

$$\begin{array}{lcl} hyp_1 & : & true(\boxed{\underline{a^+}_{\{c_1\}} \supset b}_{\{c_1\}})^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ \\ & \vdash & true(\boxed{\underline{\neg\underline{a^-}_{\{c_1\}}}}_{\{c_1\}})^+ \end{array}$$

and then with rule **wr-**$\neg_i$:

$$\begin{array}{lcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ \\ & \vdash & \boxed{\underline{true(a)^-}_{\{c_1\}} \rightarrow true(\bot)^+}_{\{c_1\}} \end{array}$$

Weak-fertilise is the next applicable method; it refines the goal with hypothesis hyp_4. When this has been done, the disjunction has been recovered from hyp_4 but there is a

new hypothesis hyp_5:

$$\begin{array}{lcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ hyp_5 & : & true(a)^- \\ \\ & \vdash & true((\neg a^-)^+ \vee b)^+ \end{array} \tag{9.9}$$

D-reduction will give the same annotation as in sequent 9.8 having also to backtrack to a weaker annotation. If the annotation that places a in the skeleton is selected, **reduce-goal** and **weak-fertilise** will perform as before and will lead to a sequent similar 9.9 only with hyp_3 repeated as hyp_6. The right annotation is then:

$$\begin{array}{lcl} hyp_1 & : & true(\underline{\boxed{a^+ \supset b^-_{\{c_2\}}}}_{\{c_2\}})^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ hyp_5 & : & true(a)^- \\ \\ & \vdash & true(\underline{\boxed{(\neg a^-)^+ \vee b^+_{\{c_2\}}}}_{\{c_2\}})^+ \end{array}$$

The goal is now rippled with rule **wr-**$\vee_{ir}$:

$$\begin{array}{lcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+) \rightarrow true(\bot)^- \\ hyp_5 & : & true(a)^- \\ \\ & \vdash & true(b)^+ \end{array}$$

This enables weak-fertilisation with hyp_2 and an axiom is obtained:

$$\begin{array}{lcl} hyp_1 & : & true(a^+ \supset b^-)^- \\ hyp_2 & : & true(a)^+ \rightarrow true(b)^- \\ hyp_3 & : & true(\neg((\neg a^-)^+ \vee b^+)^+)^- \\ hyp_4 & : & true((\neg a^-)^+ \vee b^+)^+ \rightarrow true(\bot)^- \\ hyp_5 & : & true(a)^- \\ \\ & \vdash & true(a)^+ \end{array}$$

9.3.6 Discussion

In classical propositional logic, we find rewrite rule **rw-*raa*** which is used often and is not a wave rule because it is not measure decreasing.

In the example we find that method **unblock** applies this rule when difference reduction is not possible. In classical logic, in contrast with rewrite **rw-⊥** in intuitionistic logic, rule **rw-*raa*** does not enable weak-fertilisation after it is applied. In fact, in the example, method **unblock** applies two consecutive times before another method is applicable. Since the application of this rule does not occur when **unblock** looks for rules that enable weak-fertilisation, other rewrite rules may be applicable at the same time and the planner may be diverted into unprovable branches.

In Sequent Calculus presentations of logics [Gentzen 69] as well as in Tableaux systems [Smullyan 68][Gallier 86], there is a complete search strategy for classical propositional logic by applying rules exhaustively and making connections whenever possible. Our system here, has a more directed approach because redundancy is avoided by applying wave rules only to expressions annotated and therefore already selected as potential connections. If rippling succeeds in isolating the connecting expressions, the effort involved in terms of redundant rule applications is far less than it would be for Sequent Calculus presentations and Tableaux systems where there is a more redundancy.

If, however, the rippling heuristic is not effective in isolating some connections, then the search introduced by backtracking will probably be just as expensive as the brute force approach of Sequent and Tableaux systems. We still have no proof that our method is complete for this logic.

9.4 Predicate Logic

In Predicate Logic we worry about objects and their properties. This distinction introduces new challenges to the system. The internalisation of the logic in LF represents predicates as functions from a type of objects into the type of formulae.

The problem this representation brings to our system is that, in order to find con-

nections, a distinction between object level constants and variables needs to be made. For example, $P(a)$ and $\lambda_{x:o}P(x)$ represent a potential connection but the difference unifier is first order and can only include functions in the skeleton if their arguments are d-unifiable.

What is needed is a higher-order difference unification algorithm to obtain something like:

$$\underline{\boxed{\underline{P_{\{c_1\}}}(a)}}_{\{c_1\}} \equiv_{du} \underline{\boxed{\lambda_{x:o}\underline{P_{\{c_1\}}}(x)}}_{\{c_1\}}$$

Unfortunately this algorithm is not available at the moment of writing.

9.4.1 Inference Rules

To obtain a N.D. presentation of predicate logic we add the following rules to any of the previous propositional logics depending on whether we want a minimal, intuitionistic or classical Predicate Calculus.

$$\frac{A(u)}{\forall x.A(x)}\ (\forall_i) \qquad \frac{\forall x.A(x)}{A(t)}\ (\forall_e)$$

$$\frac{A(t)}{\exists x.A(x)}\ (\exists_i) \qquad \frac{\exists x.A(x) \qquad \begin{array}{c}[A(u)]\\ \vdots \\ C\end{array}}{C}\ (\exists_e)$$

where u must not appear free in the context (an *eigenvariable*).

These rules are represented in LF as:

$$\forall I : \Pi_{p:i\to o}(\Pi_{t:i} true(p(t))) \to true(\forall(p)) \tag{9.10}$$

$$\forall E : \Pi_{p:i\to o}\Pi_{t:i} true(\forall(p)) \to true(p(t))$$

$$\exists I : \Pi_{p:i\to o}\Pi_{t:i} true(p(t)) \to true(\exists(p)) \tag{9.11}$$

$$\exists E : \Pi_{p:i\to o}\Pi_{q:o} true(\exists(p)) \to (\Pi_{t:i} true(p(t)) \to true(q)) \to true(q)$$

9.4.2 Rewrite Rules

The rewrite rules we obtain from this signature are listed below. One important difference between these rules and the ones we have been using earlier is that we have

improper rules (Section 4.2.3). Such rules introduce meta-variables into the sequent when applied.

$$true(\forall P^+)^+ \Longrightarrow \Pi_{t:i} true(P^+(t))^+ \qquad (rw{-}\forall_i)$$

$$true(\forall P^-)^- \Longrightarrow true(P^-(T))^- \qquad (rw{-}\forall_e)$$

$$true(\exists P^+)^+ \Longrightarrow true(P^+(T))^+ \qquad (rw{-}\exists_i)$$

$$true(\exists P^-)^- \Longrightarrow true(P^-(\hat{t}))^- \qquad (rw{-}\exists_e)$$

The last rule includes the symbol $\hat{t}$ which denotes a new constant of the right type in the context; it is a *context rule* (Section 4.2.3).

9.4.3 Wave Rules

Wave rules obtained from these rewrites may also introduce meta-variables. Since we are now using an explicit function application operator, as described above, coloured difference unification is capable of annotating function names independently from the terms they are applied to.

$$true(\boxed{\underline{\forall P^+}_{C_1}}_{C_1})^+ \Longrightarrow \boxed{\Pi_{t:i} true(\boxed{\underline{P^+}_{C_1}(t)}_{C_1})^+}_{C_1} \qquad (wr{-}\forall_i)$$

$$true(\boxed{\underline{\forall P^-}_{C_1}}_{C_1})^- \Longrightarrow true(\boxed{\underline{P^-}_{C_1}(T)}_{C_1})^- \qquad (wr{-}\forall_e)$$

$$true(\boxed{\underline{\exists P^+}_{C_1}}_{C_1})^+ \Longrightarrow true(\boxed{\underline{P^+}_{C_1}(T)}_{C_1})^+ \qquad (wr{-}\exists_i)$$

$$true(\boxed{\underline{\exists P^-}_{C_1}}_{C_1})^- \Longrightarrow \boxed{true(\boxed{\underline{P^-}_{C_1}(\hat{t})}_{C_1})^-}_{C_1} \qquad (wr{-}\exists_e)$$

9.4.4 Search in Predicate Logic

In predicate logic we have two new types of wave rule: improper wave rules and context wave rules.

For predicate logic we need higher-order difference unification. Since arguments of predicates vary, the skeletons of the annotations usually involve the predicates. Difference

reduction in predicate logic ripples annotations to make connections on the predicates. This is often possible because the wave rules introduce meta variables as arguments of the predicates and these meta-variables are instantiated when methods **axiom** and **weak-fertilise** are applied. These instantiations may introduce some problems as we will see in the examples.

Using higher-order difference unification, however, increases considerably the number of d-unifications between two terms. For example, the terms $true(a \wedge b)$ and $true(a \wedge c)$ return the following annotations:

$$\boxed{true(\underline{a}_{\{c_1\}} \wedge b)} \equiv_{du} \boxed{true(\underline{a}_{\{c_1\}} \wedge c)} \tag{9.12}$$

$$true(\boxed{\underline{a}_{\{c_1\}} \wedge b}) \equiv_{du} true(\boxed{\underline{a}_{\{c_1\}} \wedge c}) \tag{9.13}$$

$$\boxed{\underline{true}_{\{c_1\}}(a \wedge b)} \equiv_{du} \boxed{\underline{true}_{\{c_1\}}(a \wedge c)} \tag{9.14}$$

$$true(\boxed{a \underline{\wedge}_{\{c_1\}} b}) \equiv_{du} true(\boxed{a \underline{\wedge}_{\{c_1\}} c}) \tag{9.15}$$

From all these annotations, only 9.12 is useful with the rules we have discussed so far. An annotation where the judgement name is not in the skeleton like 9.13 would be useful in logics where there are many judgements and there are rules that link them. An example of such logic is $S4$ as presented in [Avron *et al* 87]. There, judgements *taut* and *valid* are used to separate the rules for the modal connectives from those for classical propositional logic. There are rules that link the two judgements like:

$$\Pi_{a:o} taut(a) \rightarrow valid(a)$$

A rule like this could be converted into a wave rule if an order is imposed on the judgements that would make the rule measure decreasing:

$$\boxed{valid(\underline{A}_{\{c_1\}})} \Longrightarrow \boxed{taut(\underline{A}_{\{c_1\}})}$$

The annotations in 9.14 and 9.15 are not useful in the logics we are analysing but augment the search for annotations in the algorithm. Difference unification could be modified to incorporate some restrictions on the structure of the terms. For instance a restriction could be that the judgement name be included in the skeleton or that all terms in the skeleton must be fully applied. Restrictions like this in the algorithm

could prone the search space of annotations, reduce the number of solutions and save some effort from the **d-reduction** method to try to find applicable wave rules to the annotations.

9.4.5 Examples

Example 7 $true(\forall p \supset \exists p)$

Start with **Balance** and d-unification:

$$\begin{array}{rcl} hyp_1 & : & true(\underline{\underline{\boxed{\forall p^-_{\{c_1\}}}}}_{\{c_1\}}\ )^- \\ \vdash & & true(\underline{\underline{\boxed{\exists p^+_{\{c_1\}}}}}_{\{c_1\}}\ )^+ \end{array}$$

Ripple the hypothesis:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(\underline{\underline{\boxed{p^-_{\{c_1\}}(T_1)}}}_{\{c_1\}}\ )^- \\ \vdash & & true(\underline{\underline{\boxed{\exists p^+_{\{c_1\}}}}}_{\{c_1\}}\ )^+ \end{array}$$

Ripple the goal:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(\underline{\underline{\boxed{p^-_{\{c_1\}}(T_1)}}}_{\{c_1\}}\ )^- \\ \vdash & & true(\underline{\underline{\boxed{p^+_{\{c_1\}}(T_2)}}}_{\{c_1\}}\ )^+ \end{array}$$

At this point, both the goal and the hypotheses are fully rippled; all is needed now is to instantiate meta-variables T_1 and T_2 to equal terms.

This example is only provable if there is a constant of type i in the context. This feature is reflected in the sequent above in the need of a common term. At the planning level, this sequent would be deemed to be an axiom because well-typedness is not checked at this level. When the plan is executed at the object level by the proof editor however, it will need to instantiate the meta-variables to a common term if there is one available or if there is a choice operator (c.f. [Avron *et al* 87]).

Example 8 $true(\forall \lambda_{x:i}.(p(x) \supset q(x)) \supset \forall p \supset \forall q)$

This example uses similar techniques to the previous one but with two colours.

Start with Balance and d-unification:

$$\begin{array}{lcl} hyp_1 & : & true(\boxed{\forall\lambda_{x:i}.(\underline{p^+_{\{c_1\}}}(x) \supset \underline{q^-_{\{c_2\}}}(x))}_{\{c_1,c_2\}})^- \\ \vdash & & true(\boxed{\boxed{(\forall\underline{p^-_{\{c_1\}}})^-}_{\{c_1\}} \supset \boxed{(\forall\underline{q^+_{\{c_2\}}})^+}_{\{c_2\}}}_{\{c_1,c_2\}})^+ \end{array}$$

Ripple the hypothesis; first with rule **wr-$\forall_e$**:

$$\begin{array}{lcl} hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\ hyp_2 & : & true(\boxed{\underline{p^+_{\{c_1\}}}(T) \supset \underline{q^-_{\{c_2\}}}(T)}_{\{c_1,c_2\}})^- \\ \vdash & & true(\boxed{\boxed{(\forall\underline{p^-_{\{c_1\}}})^-}_{\{c_1\}} \supset \boxed{(\forall\underline{q^+_{\{c_2\}}})^+}_{\{c_2\}}}_{\{c_1,c_2\}})^+ \end{array}$$

and then with rule **wr-$\supset_e$**:

$$\begin{array}{lcl} hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\ hyp_2 & : & true(p^+(T) \supset q^-(T))^- \\ hyp_3 & : & \boxed{true(\underline{\boxed{\underline{p^+_{\{c_1\}}}(T)}_{\{c_1\}}}_{\{c_1\}})^+ \rightarrow true(\underline{\boxed{\underline{q^-_{\{c_2\}}}(T)}_{\{c_2\}}}_{\{c_2\}})^-}_{\{c_1,c_2\}} \\ \vdash & & true(\boxed{\boxed{(\forall\underline{p^-_{\{c_1\}}})^-}_{\{c_1\}} \supset \boxed{(\forall\underline{q^+_{\{c_2\}}})^+}_{\{c_2\}}}_{\{c_1,c_2\}})^+ \end{array}$$

Ripple the goal; first with rule **wr-$\supset_i$**:

$$\begin{array}{lcl} hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\ hyp_2 & : & true(p^+(T) \supset q^-(T))^- \\ hyp_3 & : & \boxed{true(\underline{\boxed{\underline{p^+_{\{c_1\}}}(T)}_{\{c_1\}}}_{\{c_1\}})^+ \rightarrow true(\underline{\boxed{\underline{q^-_{\{c_2\}}}(T)}_{\{c_2\}}}_{\{c_2\}})^-}_{\{c_1,c_2\}} \\ \vdash & & \boxed{true(\underline{\boxed{\forall\underline{p^-_{\{c_1\}}}}_{\{c_1\}}}_{\{c_1\}}) \rightarrow true(\underline{\boxed{\forall\underline{q^+_{\{c_2\}}}}_{\{c_2\}}}_{\{c_2\}})^+}_{\{c_1,c_2\}} \end{array}$$

then with rule **wr-$\forall_e$**:

$$\begin{array}{lcl} hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\ hyp_2 & : & true(p^+(T) \supset q^-(T))^- \\ hyp_3 & : & \boxed{true(\underline{\boxed{\underline{p^+_{\{c_1\}}}(T)}_{\{c_1\}}}_{\{c_1\}})^+ \rightarrow true(\underline{\boxed{\underline{q^-_{\{c_2\}}}(T)}_{\{c_2\}}}_{\{c_2\}})^-}_{\{c_1,c_2\}} \\ \vdash & & \boxed{true(\underline{\boxed{\underline{p^-_{\{c_1\}}}(T_1)}_{\{c_1\}}}_{\{c_1\}})^- \rightarrow true(\underline{\boxed{\forall\underline{q}_{\{c_2\}}}_{\{c_2\}}}_{\{c_2\}})}_{\{c_1,c_2\}} \end{array}$$

and finally rule **wr-**$\forall_i$:

$$
\begin{array}{lll}
hyp_1 & : & true(\forall\lambda_{x:i}.(p(x) \supset q(x))) \\
hyp_2 & : & true(p^+(T) \supset q^-(T))^- \\
hyp_3 & : & \boxed{true(\underline{\underline{\boxed{p^+_{\{c_1\}}(T)}}}_{\{c_1\}})^+ \rightarrow true(\underline{\underline{\boxed{q^-_{\{c_2\}}(T)}}}_{\{c_2\}})^-}_{\{c_1,c_2\}} \\
\vdash & & \boxed{true(\underline{\underline{\boxed{p^-_{\{c_1\}}(T_1)}}}_{\{c_1\}})^- \rightarrow \underline{\underline{\boxed{\Pi_{t:i} true(q^+_{\{c_2\}}(t))^+}}}_{\{c_2\}}}_{\{c_1,c_2\}}
\end{array}
$$

Weak-fertilising with hyp_3:

$$
\begin{array}{lll}
hyp_1 & : & true(\forall\lambda_{x:i}.(p(x) \supset q(x))) \\
hyp_2 & : & true(p^+(T) \supset q^-(T))^- \\
hyp_3 & : & \boxed{true(\underline{\underline{\boxed{p^+_{\{c_1\}}(T)}}}_{\{c_1\}})^+ \rightarrow true(\underline{\underline{\boxed{q^-_{\{c_2\}}(T)}}}_{\{c_2\}})^-}_{\{c_1,c_2\}} \\
hyp_4 & : & true(p(T_1))^- \\
\vdash & & true(p(\hat{t}))^+
\end{array}
$$

Making meta-variable T_1 equal to $\hat{t}$ we get an axiom.

The process just described builds a full proof plan for the theorem but when it is passed to the proof editor, the plan does not succeed. The reason for this is that the constant $\hat{t}$ which instantiates meta-variable T_1 is introduced in the last sequent above after hyp_4 enters the context and so the instantiated term $true(p(\hat{t}))$ does not type check in the object proof.

The planner is capable of producing the right proof plan by using *eager* **balance** instead. Using this strategy, the sequent looks as follows (after balancing and annotating):

$$
\begin{array}{lll}
hyp_1 & : & true(\underline{\boxed{\forall\lambda_{x:i}.(p^+(x) \supset \underline{\underline{\boxed{q^-_{\{c_1\}}(x)}}}_{\{c_1\}})}}_{\{c_1\}})^- \\
hyp_2 & : & true(\forall p^-))^- \\
\vdash & & true(\underline{\underline{\boxed{q^+_{\{c_1\}}(\hat{t})}}}_{\{c_1\}})^+
\end{array}
$$

Here, the constant $\hat{t}$ was introduced after hyp_2. Difference unification would identify a connection between the goal and hyp_1. Rippling q in hyp_1 is similar to the way it was done above using wave **wr-**$\supset_e$:

$$
\begin{array}{lcl}
hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\
hyp_2 & : & true(\forall p^-)^- \\
hyp_3 & : & \boxed{true(p(T))^+ \rightarrow true(\underline{\underline{\boxed{q^-_{\{c_1\}}(T)}}}_{\{c_1\}}\ )^-}_{\{c_1\}} \\
\vdash & & true(\underline{\underline{\boxed{q^+_{\{c_1\}}(\hat{t})}}}_{\{c_1\}}\ )^+
\end{array}
$$

Weak-fertilise applies with hypothesis hyp_3 refining the goal as follows:

$$
\begin{array}{lcl}
hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\
hyp_2 & : & true(\forall p^-)^- \\
hyp_3 & : & true(p(T))^+ \rightarrow true(q(T))^- \\
\vdash & & true(p(\hat{t}))^+
\end{array}
$$

Method **d-reduction** d-unifies again:

$$
\begin{array}{lcl}
hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\
hyp_2 & : & true(\underline{\underline{\boxed{\forall p^-_{\{c_2\}}}}}_{\{c_2\}}\ )^- \\
hyp_3 & : & true(p(T))^+ \rightarrow true(q(T))^- \\
\vdash & & true(\underline{\underline{\boxed{p_{\{c_2\}}(\hat{t})}}}_{\{c_2\}}\ )^+
\end{array}
$$

Now, rippling hyp_2 with rule **wr-**$\forall_e$ produces:

$$
\begin{array}{lcl}
hyp_1 & : & true(\forall\lambda_{x:i}.(p^+(x) \supset q^-(x)))^- \\
hyp_2 & : & true(\forall p^-)^- \\
hyp_3 & : & true(p(T))^+ \rightarrow true(q(T))^- \\
hyp_4 & : & true(p(T_1))^- \\
\vdash & & true(\underline{\underline{\boxed{p_{\{c_2\}}(\hat{t})}}}_{\{c_2\}}\ )^+
\end{array}
$$

This time, meta-variable T_1 appears after term $\hat{t}$ is introduced so they can be unified by method **axiom** and the plan will be executable at the object level.

Example 9 $true(\forall p \supset \neg\exists\neg p)$

This example combines negation and quantifiers.

Balance:

$$\begin{array}{rcl} hyp_1 & : & true(\boxed{\underline{\forall p^-}_{\{c_1\}}}^{\{c_1\}}\ )^- \\ \vdash & & true(\boxed{\underline{\neg\exists\neg p^+}_{\{c_1\}}}^{\{c_1\}}\ )^+ \end{array}$$

Ripple the hypothesis with rule **wr-$\forall_e$**:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(\boxed{\underline{p^-}_{\{c_1\}}(T)}^{\{c_1\}}\ )^- \\ \vdash & & true(\boxed{\underline{\neg\exists\neg p^+}_{\{c_1\}}}^{\{c_1\}}\ )^+ \end{array}$$

Ripple the goal; first with rule **wr-$\neg_i$**:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(\boxed{\underline{p}_{\{c_1\}}(T)}^{\{c_1\}}\ )^- \\ \vdash & & \boxed{true(\boxed{\underline{\exists\neg p^+}_{\{c_1\}}}^{\{c_1\}}\ )^- \underline{\qquad}_{\{c_1\}} \rightarrow true(\bot)^+}^{\{c_1\}} \end{array}$$

then rule **wr-$\exists_e$**:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(\boxed{\underline{p^+}_{\{c_1\}}(T)}^{\{c_1\}}\ )^- \\ \vdash & & \boxed{true(\boxed{\underline{\neg p^+}_{\{c_1\}}(\hat{t})}^{\{c_1\}}\ )^- \underline{\qquad}_{\{c_1\}} \rightarrow true(\bot)^-}^{\{c_1\}} \end{array}$$

and rule **wr-$\neg_e$**:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(\boxed{\underline{p^-}_{\{c_1\}}(T)}^{\{c_1\}}\ )^- \\ \vdash & & \boxed{\underline{(true(p^+(\hat{t}))^+ \rightarrow true(\bot)^-)^-}_{\{c_1\}} \rightarrow true(\bot)^+}^{\{c_1\}} \end{array}$$

Balance introduces the $\rightarrow$:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(p(T))^- \\ hyp_3 & : & true(p(\hat{t}))^+ \rightarrow true(\bot)^- \\ & \vdash & true(\bot)^+ \end{array}$$

Weak fertilising with hyp_3:

$$\begin{array}{rcl} hyp_1 & : & true(\forall p^-)^- \\ hyp_2 & : & true(p(T))^- \\ hyp_3 & : & true(p(\hat{t}))^+ \rightarrow true(\bot)^- \\ & \vdash & true(p(\hat{t}))^+ \end{array}$$

Method **Axiom** tries to unify T and $\hat{t}$ but, as before, this would not type-check in the object level proof because $\hat{t}$ was introduced after hyp_2. If the planner fails at this sequent, it would backtrack to the point where the hypotheses are rippled and will try to ripple the goal first. This would introduce $\hat{t}$ before hyp_1 is eliminated and the problem would be solved.

9.4.6 Discussion

Predicate logic rules introduce new aspects to search. Context rules introduce new object-level terms wherever they are applied. Rules **wr-**$\forall_e$ and **wr-**$\exists_i$ introduce meta variables into terms that are instantiated later on in the proof.

The order in which the object-level terms and the meta-variables are introduced is important and may cause the final plan no to be applicable at the object level. To avoid this problem the first plan, it is necessary to build into the system a mechanism to keep track of the introduction of new constants so that the methods can verify that the instantiations are done in the right order.

9.5 Modal Logics

For these section we use the N.D. presentation of Modal logics in [Simpson 94]. The possible-world semantics is incorporated into the standard N.D. treatment of propositional logic to come up with a uniform system that deals with a large number of modal logics. We're not going to give details but only present what is needed here.

9.5.1 Inference Rules

The basic system for modal logic K has the following introduction and elimination rules for the modal connectives $\Box$ and $\Diamond$:

$$\begin{array}{c} [xRy] \\ \vdots \\ \dfrac{y : A}{x : \Box A}\ (\Box_i)^1 \end{array} \qquad \frac{x : \Box A \quad xRy}{y : A}\ (\Box_e)$$

$$\frac{y : A \quad xRy}{x : \Diamond A}\ (\Diamond_i) \qquad \frac{x : \Diamond A \qquad \begin{array}{c} [y : A]\ [xRy] \\ \vdots \\ z : B \end{array}}{z : B}\ (\Diamond_e)^2$$

Provisos:

1. y must be different from x and must not occur in any assumptions other than the distinguished occurrences of xRy

2. y must be different from both x and z and must not occur in any open assumption upon which $z : B$ depends other than the distinguished occurrences of $y : A$ and xRy

$x : A$ means "A is true in world x" and R is the world-visibility relation.

The rules for $\Box$ and $\Diamond$ are added to an version of the rules for propositional logic (Sections 9.1.1,9.2.1 and 9.3.1) that have the judgements extended to include world constants. The LF encoding of these rules is the following[1]:

[1] The Propositional part of the signature is intuitionistic.

$$
\begin{array}{rcl}
o, W & : & Type \\
true & : & w \rightarrow o \rightarrow Type \\
\bot & : & o \\
\wedge, \vee, \supset & : & o \rightarrow o \rightarrow o \\
\Box, \Diamond & : & o \rightarrow o
\end{array}
$$

$$
\begin{array}{rcl}
\wedge_i & : & \Pi_{A,B:o}\Pi_{x:W} true(x,A) \rightarrow true(x,B) \rightarrow true(x, A \wedge B) \\
\wedge_{el} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A \wedge B) \rightarrow true(x, A) \\
\wedge_{er} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A \wedge B) \rightarrow true(x, B) \\
\vee_{il} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A) \rightarrow true(x, A \vee B) \\
\vee_{ir} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, B) \rightarrow true(x, A \vee B) \\
\vee_e & : & \Pi_{A,B:o}\Pi_{x,y:W} true(x, A \vee B) \rightarrow \Pi_{C:o}((true(x,A) \rightarrow true(y,C)) \\
 & & \quad \rightarrow (true(x,B) \rightarrow true(y,C)) \rightarrow true(y,C)) \\
\supset_i & : & \Pi_{A,B:o}\Pi_{x:W}(true(x,A) \rightarrow true(x,B)) \rightarrow true(x, A \supset B) \\
\supset_e & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A \supset B) \rightarrow true(x,A) \rightarrow true(x,B) \\
\neg_i & : & \Pi_{A:o}\Pi_{x:W}(true(x,A) \rightarrow true(x,\bot)) \rightarrow true(x, \neg A) \\
abs & : & \Pi_{A:o}\Pi_{x,y:W} true(x,\bot) \rightarrow true(y,A) \\
\Box_i & : & \Pi_{A:o}\Pi_{x:W}(\Pi_{y:W}\ xRy \rightarrow true(y,A)) \rightarrow true(x, \Box A) \\
\Box_e & : & \Pi_{A:o}\Pi_{x:W}\ true(x, \Box A) \rightarrow xRy \rightarrow true(y,A) \\
\Diamond_i & : & \Pi_{A:o}\Pi_{x,y:W}\ true(y,A) \rightarrow xRy \rightarrow true(x, \Diamond A) \\
\Diamond_e & : & \Pi_{A,B:o}\Pi_{x,z:W}\ true(x, \Diamond A) \rightarrow (\Pi_{y:W}\ true(y,A) \rightarrow xRy \rightarrow true(z,B)) \rightarrow true(z,B)
\end{array}
$$

The judgement $true(x, A)$ in this signature means "proposition A is true in world x".

9.5.2 Restrictions on the Visibility Relation

In the modal logic K sketched above there is no particular requirement for the world visibility relation. It is by imposing restrictions on this relation —as is well known— that we obtain different modal logics. In the system we use here, these restrictions are incorporated into the formal system in the form of (N.D. style) rules:

$\forall x.\exists y.xRy$ translates into:

$$\dfrac{\begin{array}{c}[xRy] \\ \vdots \\ z : A\end{array}}{z : A} \qquad (D)$$

$\forall x.xRx$ translates into:

$$\dfrac{\begin{array}{c}[xRx] \\ \vdots \\ z : A\end{array}}{z : A} \qquad (T)$$

$\forall xy.xRy \supset yRx$ translates into:

$$\dfrac{\begin{array}{cc} & [yRx] \\ & \vdots \\ xRy & z : A \end{array}}{z : A} \qquad (B)$$

$\forall xyz.xRy \wedge yRz \supset xRz$ translates into:

$$\dfrac{\begin{array}{ccc} & & [xRz] \\ & & \vdots \\ xRy & yRz & w : A \end{array}}{w : A} \qquad (4)$$

$\forall xyz.xRy \wedge xRz \supset xRz$ translates into:

$$\dfrac{\begin{array}{ccc} & & [yRz] \\ & & \vdots \\ xRy & xRz & w : A \end{array}}{w : A} \qquad (5)$$

$\forall xyz.xRy \wedge xRz \supset \exists w.yRw \wedge zRw$ translates into:

$$\dfrac{\begin{array}{ccc} & & [yRw][zRw] \\ & & \vdots \\ xRy & xRz & v : A \end{array}}{v : A} \qquad (2)$$

In rule **D**, y must be different from both x and z and must not occur in any assumption other than the distinguished occurrences of xRy.

In rule **2**, w must be different from x,y,z,v and must not occur in any open assumptions other than the distinguished occurrences of yRw and zRw.

These N.D. rules are represented in LF as follows:

$$\begin{array}{rcl}
D & : & \Pi_{A:o}\Pi_{x,y,z:W}(xRy \rightarrow true(z,A)) \rightarrow true(z,A) \\
T & : & \Pi_{A:o}\Pi_{x,z:W}(xRx \rightarrow true(z,A)) \rightarrow true(z,A) \\
B & : & \Pi_{A:o}\Pi_{x,y,z:W}xRy \rightarrow (yRx \rightarrow true(z,A)) \rightarrow true(z,A) \\
4 & : & \Pi_{A:o}\Pi_{x,y,z,w:W}xRy \rightarrow yRz \rightarrow (xRz \rightarrow true(w,A)) \rightarrow true(w,A) \\
5 & : & \Pi_{A:o}\Pi_{x,y,z,w:W}xRy \rightarrow xRz \rightarrow (yRz \rightarrow true(w,A)) \rightarrow true(w,A) \\
2 & : & \Pi_{A:o}\Pi_{x,y,z,w:W}xRy \rightarrow xRz \rightarrow (yRw \rightarrow zRw \rightarrow true(w,A)) \rightarrow true(w,A)
\end{array}$$

Adding the appropriate axioms from the list above to the signature given in the last section results in signatures for different modal logics. See [Hughes & Cresswell 90] for a description of the systems that can be obtained with these axioms.

9.5.3 Rewrite Rules

The rewrite rules for the propositional part of the signature are extensions to the rewrites described for Propositional logic. These extensions contain an extra argument in every atomic judgement according to the inference rules in section 9.5.1. We don't include these rules here but only those for the modal connectives:

$$true(x, \Box A^+)^+ \Longrightarrow (\Pi_{y:W}\ (xRy)^- \rightarrow true(y, A)^+)^+ \qquad (rw{-}\Box_i)$$

$$true(x, \Box A^-)^- \Longrightarrow (\Pi_{y:W}\ (xRy)^+ \rightarrow true(y, A)^-)^- \qquad (rw{-}\Box_e)$$

$$true(x, \Diamond A^+)^+ \Longrightarrow \left\{ \begin{array}{l} (xRY)^+ \\ true(Y, A)^+ \end{array} \right. \qquad (rw{-}\Diamond_i)$$

$$true(x, \Diamond A^-)^- \Longrightarrow (xR\hat{y})^- \times true(\hat{y}, A)^- \qquad (rw{-}\Diamond_e)$$

In **rw-**$\Diamond_i$, Y is a new meta-variable as in rules **rw-**$\forall_e$ and **rw-**$\exists_i$ in Predicate Calculus (Section 9.4.1).

Rule **rw-**$\Diamond_e$ is a type of rewrite rule we have not used before. The effect of applying rule $\Diamond_i$ backwards in a proof is to introduce two new assumptions in the hypothesis list: xRy and $true(y', A)$. $\hat{y}$ is a new term as in rule **rw-**$\exists_e$ in Predicate Calculus. This rule is similar to the sequent calculus $\wedge$-left rule:

$$\frac{a, b \vdash C}{a \wedge b \vdash C}$$

The rewrite **rw-**$\Diamond_e$ involves the symbol $\times$ to indicate that its arguments are to be included as two new hypotheses. The rewriting mechanism used in the methods and submethods does the appropriate rewriting when this symbol is involved in the rule.

The constant $\hat{y}$ in **rw-**$\Diamond_e$ is another variation of this rule with respect to the rest. This LF constant is introduced by the rule application as a new object level variable with respect to the context of the sequent where the rewriting is done. This is also done by the rewriting mechanism of the methods and submethods.

The translation of the rules for the restrictions on the world-visibility relation presents some problems. First, the rules for axiom D and T can only be written as right-to-left rules because the term involving R occurs within a function type in the antecedent

of the rule. The right-to-left version, however, is not useful because it is diverging (it introduces terms) and is not constrained (it would apply to any term of the form $true(z, A)$. The rules for D and T are:

$$true(Z, A)^- \Longrightarrow ((_X R_Y)^+ \rightarrow true(Z, A)^-)^- \qquad (rw{-}D)$$

$$true(Z, A)^- \Longrightarrow ((_X R_X)^+ \rightarrow true(Z, A)^-)^- \qquad (rw{-}D)$$

The rest of the rules can be translated as:

$$(_X R_Y)^- \Longrightarrow ((_Y R_X)^- \rightarrow true(Z, A)^+)^+ \rightarrow true(Z, A)^- \qquad (rw{-}B)$$

$$(_X R_Y)^- \Longrightarrow (_Y R_Z)^+ \rightarrow ((_X R_Z)^- \rightarrow true(W, A)^+)^+ \rightarrow true(W, A)^- \qquad (rw{-}4)$$

$$(_X R_Y)^- \Longrightarrow (^+_X R_Z)^+ \rightarrow ((_Y R_Z)^- \rightarrow true(W, A)^+)^+ \rightarrow true(W, A)^- \qquad (rw{-}5)$$

$$(_X R_Y)^- \Longrightarrow (_X R_Z)^+ \rightarrow ((_Y R_W)^- \rightarrow (_Z R_W)^- \rightarrow true(W, A)^+)^+ \rightarrow true(W, A)^- \qquad (rw{-}2)$$

The place-holder terms $true(W, A)$ cannot be removed from these rules as we've done before. The reason for this is that these terms are not place-holders for all possible goals in the sequents. The rewriting tactic would not be able to carry out the object level operations corresponding to the rewriting if the goal of the sequent at the time is of the form xRy for instance.

9.5.4 Wave Rules

The wave rules we present in this section differ from those we have used before in that to annotate them, we assume that the difference unifier used to compare both sides of the rewrite rules provide relational annotations. That is, the skeletons of the expressions compared may be related by a relation other than equality (R in this case).

Since the rewrite rules for the world-visibility relation described in the last section cannot be parsed into wave rules (they would not be measure decreasing), we list only the wave rules for the modal connectives:

$$true(x, \underline{\boxed{\Box \underline{A^+}_{C_1}}}_{C_1})^+ \Longrightarrow \underline{\boxed{(\Pi_{y:W}\ (xRy)^- \rightarrow \underline{true(y, A)^+}_{C_1})^+}}_{C_1} \qquad (wr{-}\Box_i)$$

$$true(x, \underline{\boxed{\underline{\Box A^-}_{C_1}}}_{C_1})^- \Longrightarrow \underline{\boxed{\Pi_{y:W}\ (xRy)^+ \rightarrow \underline{true(y, A^-)^-}_{C_1}}}_{C_1} \qquad (wr{-}\Box_e)$$

$$true(x, \underline{\boxed{\underline{\Diamond A^+}_{C_1}}}_{C_1})^+ \Longrightarrow \begin{cases} (xR_Y)^+ \\ \underline{true(Y, A^+)^+}_{C_1} \end{cases} \qquad (wr{-}\Diamond_i)$$

$$true(x, \underline{\boxed{\underline{\Diamond A^-}_{C_1}}}_{C_1})^- \Longrightarrow (xR\hat{y})^- \times \underline{true(\hat{y}, A^-)^-}_{C_1} \qquad (wr{-}\Diamond_e)$$

The symbols $\hat{y}$ and $\times$ have the same roles they had in the rewrite rules we described in the previous section.

9.5.5 Search in Modal Logics

In modal logic K, where there is no restriction on the world visibility relation, the search process followed by the planner is similar to that of the previous logics. Rule **wr-**$\Diamond_i$ is an improper rule and rule **wr-**$\Diamond_e$ is a context rule. These two kinds of rule where used in predicate logic before (Section 9.4). The new symbol in this logic is $\times$ but it does not introduce any problem as far as proof search is concerned because it only introduces two new hypotheses simultaneously in the context.

Modal logics where world visibility constraints apply, however, do get affected by the corresponding rules. The original N.D. system in [Simpson 94] is designed to use the world-visibility relations as assumptions only and hence, in none of the rules this relation appears in the goal. Our translation of the rules for the modal connectives produces rules that introduce terms involving R as goals (Let's call terms involving R, R-terms). These goals are not solvable by either of the rules because:

1. There are no wave rules for R-terms.
2. Rewrites **rw-**D and **rw-**T are not applicable to R-terms. The rest of the rewrites apply to terms R-terms in the context but all the new terms don't have R-terms as heads so they would not weak-fertilise an R-term in the goal.
3. Inference rules have non-R-terms in their heads and, as with rewrite rules, they cannot be used to reduce an R-term in the goal.

These considerations imply that the method **unblock** will not be able to solve sequents with R-terms in the goal, only method **axiom** will solve them if they are trivial. The last case applies for modal logic K as we will see.

For the other modal logics, where the world-visibility relations are needed, we don't have a way of solving sequents with R-terms in the goal. As we will see in the examples, however, when running the planner with the methods as they currently are, all leaves contain a sequent which is an instance of the corresponding world visibility relation.

This indicates that our system is incomplete for logics with this kind of rule but still solves the main part of the proof. The remaining sequents could be solved by a symbolic evaluation method that verifies that the sequent is effectively an instance of one of the assumptions.

9.5.6 Examples

Example 10 $true(x, \Box(a \supset b) \supset \Diamond a \supset \Diamond b)$

The planner starts by applying **balance**. **D-reduction** selects the following annotation:

$$\begin{array}{rcl} hyp_1 & : & true(x, \boxed{\Box(\underline{a^+}_{\{c_1\}} \supset \underline{b^-}_{\{c_2\}})}_{\{c_1,c_2\}})^- \\ \vdash & & true(x, \boxed{\Diamond\underline{a^-}_{\{c_1\}} \supset \Diamond\underline{b^+}_{\{c_2\}}}_{\{c_1,c_2\}})^+ \end{array}$$

Reduce-hyp ripples hyp_1 with rules **wr-**$\Box_e$ and **wr-**$\supset_e$:

$$\begin{array}{rcl} hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\ hyp_2 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y, a^+ \supset b^-)^- \\ hyp_3 & : & \boxed{\Pi_{y:W}\ (xRy)^+ \rightarrow \boxed{\underline{true(y,a)^+}_{\{c_1\}} \rightarrow \underline{true(y,b)^-}_{\{c_2\}}}_{\{c_1,c_2\}}}_{\{c_1,c_2\}} \\ \vdash & & true(x, \boxed{(\Diamond\underline{a^-}_{\{c_1\}})^-}_{\{c_1\}} \supset \boxed{(\Diamond(\underline{b^+}_{\{c_2\}})^+)^+}_{\{c_2\}})^+ \end{array}$$

Reduce-goal applies rule **wr-**$\supset_i$:

$$\begin{array}{rcl} hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\ hyp_2 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y, a^+ \supset b^-)^- \\ hyp_3 & : & \boxed{\Pi_{y:W}\ (xRy)^+ \rightarrow \boxed{\underline{true(y,a)^+}_{\{c_1\}} \rightarrow \underline{true(y,b)^-}_{\{c_2\}}}_{\{c_1,c_2\}}}_{\{c_1,c_2\}} \\ \vdash & & \boxed{\underline{true(x, \boxed{\Diamond\underline{a^-}_{\{c_1\}}}_{\{c_1\}})^-}_{\{c_1\}} \rightarrow \underline{true(x, \boxed{\Diamond\underline{b^+}_{\{c_2\}}}_{\{c_2\}})^+}_{\{c_2\}}}_{\{c_1,c_2\}} \end{array}$$

and then rule **wr-$\diamond_e$**:

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \Box(a_{\{c_1\}} \supset b_{\{c_2\}})) \\
hyp_2 & : & \Pi_{y:W}\ (xRy) \rightarrow true(y, a_{\{c_1\}} \supset b_{\{c_2\}}) \\
hyp_3 & : & \boxed{\Pi_{y:W}\ (xRy)^+ \rightarrow \boxed{\underline{true(y,a)^+}_{\{c_1\}} \rightarrow \underline{true(y,b)^-}_{\{c_2\}}}_{\{c_1,c_2\}}}_{\{c_1,c_2\}} \\
\vdash & & \boxed{(xR\hat{y})^- \rightarrow \underline{true(\hat{y},a)^-}_{\{c_1\}} \rightarrow true(x, \underline{\boxed{\diamond \underline{b^+}_{\{c_2\}}}_{\{c_2\}}}_{\{c_2\}})^+}_{\{c_1,c_2\}}
\end{array}
$$

Colour c_1 is fully rippled by now so we continue with c_2. Using rule **wr-$\diamond_i$** two new sequents are obtained:

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\
hyp_2 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y, a^+ \supset b^-)^- \\
hyp_3 & : & \boxed{\Pi_{y:W}\ (xRy)^+ \rightarrow \boxed{\underline{true(y,a)^+}_{\{c_1\}} \rightarrow \underline{true(y,b)^-}_{\{c_2\}}}_{\{c_1,c_2\}}}_{\{c_1,c_2\}} \\
\vdash & & \boxed{(xR\hat{y})^- \rightarrow \underline{true(\hat{y},a)^-}_{\{c_1\}} \rightarrow \underline{true(Y,b)^+}_{\{c_2\}}}_{\{c_1,c_2\}}
\end{array}
$$

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\
hyp_2 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y, a^+ \supset b^-)^- \\
hyp_3 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y,a)^+ \rightarrow true(y,b)^- \\
\vdash & & (xR\hat{y})^- \rightarrow true(\hat{y},a)^- \rightarrow (xRY_1)^+
\end{array}
$$

Method **axiom** applies to the second sequent instantiating meta-variable Y_1 with $\hat{y}$. Method **weak-fertilise** is applied to the first sequent reducing the goal with hyp_3:

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\
hyp_2 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y, a^+ \supset b^-)^- \\
hyp_3 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y,a)^+ \rightarrow true(y,b)^- \\
hyp_4 & : & (xR\hat{y})^- \\
hyp_5 & : & true(\hat{y},a)^- \\
\vdash & & true(Y_2,a)^+
\end{array}
$$

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\
hyp_2 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y, a^+ \supset b^-)^- \\
hyp_3 & : & \Pi_{y:W}\ (xRy)^+ \rightarrow true(y,a)^+ \rightarrow true(y,b)^- \\
hyp_4 & : & (xR\hat{y})^- \\
hyp_5 & : & true(\hat{y},a)^- \\
\vdash & & (xRY_2)^+
\end{array}
$$

Axiom method applies to both these sequents and instantiates meta-variable Y_2 with $\hat{y}$.

Example 11 $true(x, \Box(a \supset b) \supset (\Box a \supset \Box b))$

As usual Balance is the first step:

$$\begin{array}{rcl} hyp_1 & : & true(x, \boxed{\boxed{\Box(\underline{a^+}_{\{c_1\}} \supset \underline{b^-}_{\{c_2\}})^-}}_{\{c_1,c_2\}})^- \\ & \vdash & true(x, \boxed{\boxed{(\Box\underline{a^-}_{\{c_1\}})^- \supset (\Box\underline{b^+}_{\{c_2\}})^+}}_{\{c_1,c_2\}})^+ \end{array}$$

rippling all skeletons first in the hypotheses (rules **wr-**$\Box_e$ and **wr-**$\supset_e$) and then in the goal (rules **wr-**$\supset_i$, **wr-**$\Box_e$ and **wr-**$\Box_i$) we get:

$$\begin{array}{rcl} hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\ hyp_2 & : & \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1, a^+ \supset b^-)^- \\ hyp_3 & : & \boxed{\boxed{\Pi_{y_1:W}(xRy_1)^+ \rightarrow \underline{true(y_1,a)^+}_{\{c_1\}} \rightarrow \underline{true(y_1,b)^-}_{\{c_2\}}}}_{\{c_1,c_2\}} \\ \vdash & & \boxed{\boxed{(\Pi_{y_2:W}(xRy_2)^- \rightarrow \underline{true(y_2,a)^-}_{\{c_1\}})^- \rightarrow \Pi_{y_3:W}(xRy_3)^- \rightarrow \underline{true(y_3,b)^+}_{\{c_2\}}}}_{\{c_1,c_2\}} \end{array}$$

At this point **weak-fertilise** applies (using hyp_3):

$$\begin{array}{rcl} hyp_3 & : & \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1,a)^+ \rightarrow true(y_1,b)^- \\ hyp_4 & : & \Pi_{y_2:W}(xRy_2)^+ \rightarrow true(y_2,a)^- \\ hyp_5 & : & (xRy_3)^- \\ & \vdash & (xRy_3)^+ \end{array}$$

$$\begin{array}{rcl} hyp_3 & : & \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1,a)^+ \rightarrow true(y_1,b)^- \\ hyp_4 & : & \Pi_{y_2:W}(xRy_2)^+ \rightarrow true(y_2,a)^- \\ hyp_5 & : & (xRy_3)^- \\ & \vdash & true(y_3,a)^+ \end{array}$$

The first sequent is an axiom so method **axiom** closes the branch. The second sequent can be weak-fertilised with hyp_4 giving the following:

$$\begin{array}{rcl} hyp_1 & : & true(x, \Box(a^+ \supset b^-)^-)^- \\ hyp_2 & : & \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1, a^+ \supset b^-)^- \\ hyp_3 & : & \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1,a)^+ \rightarrow true(y_1,b)^- \\ hyp_4 & : & \Pi_{y_2:W}(xRy_2)^+ \rightarrow true(y_2,a)^- \\ hyp_5 & : & (xRy_3)^- \\ & \vdash & (xRy_3)^+ \end{array}$$

At this point, methods **axiom** closes the branch.

Example 12 $true(x, \Box a \supset a)$

This theorem is axiom T in Hilbert style presentation for modal logics. In the system we use here, logic T is obtained by assuming that R is *reflexive* (cf. restriction T in Section 9.5.1).

The first method is **balance**:

$$\begin{array}{rcl} hyp_1 & : & true(x, \boxed{(\Box \underline{a^-}_{\{c_1\}})^-}_{\{c_1\}})^- \\ & \vdash & true(x,a)^+ \end{array}$$

then ripple:

$$\begin{array}{rcl} hyp_1 & : & true(x, \Box a^-)^- \\ hyp_2 & : & \boxed{\Pi_{y:W}(xRy)^+ \rightarrow \underline{true(y,a)^-}_{\{c_1\}}}_{\{c_1\}} \\ & \vdash & true(x,a)^+ \end{array}$$

and finally weak-fertilise:

$$\begin{array}{rcl} hyp_1 & : & true(x, \Box a^-)^- \\ hyp_2 & : & \Pi_{y:W}(xRy)^+ \rightarrow true(y,a)^- \\ & \vdash & (xRx)^+ \end{array}$$

This final sequent is provable by calling upon the reflexivity restriction on the visibility relation (T).

Example 13 $true(x, a \supset \Box \diamond a)$

This theorem is axiom B in the Hilbert style presentations of modal logics. The corresponding restriction on the visibility relation is *symmetry* (cf. restriction B in Section 9.5.1).

Balance:

$$\begin{array}{rcl} hyp_1 & : & true(x,a)^- \\ & \vdash & true(x, \boxed{\Box(\diamond \underline{a^-}_{\{c_1\}})^-}_{\{c_1\}})^- \end{array}$$

Rippling goal first with rule **wr-$\Box_i$**:

$$\begin{array}{rcl} hyp_1 & : & true(x,a)^- \\ & \vdash & \boxed{\Pi_{y:W}(xRy)^- \rightarrow true(y, \boxed{\diamond \underline{a^+}_{\{c_1\}}}_{\{c_1\}})^+}_{\{c_1\}} \end{array}$$

and then with rule **wr-$\diamond_i$**:

$$\begin{array}{rcl} hyp_1 & : & true(x,a)^- \\ & \vdash & \Pi_{y:W}(xRy)^- \rightarrow (yRY)^+ \end{array}$$

$$\begin{array}{rcl} hyp_1 & : & true(x,a)^- \\ & \vdash & \boxed{\Pi_{y:W}(xRy)^- \rightarrow \underline{true(Y,a)^+}_{\{c_1\}}}_{\{c_1\}} \end{array}$$

Axiom applies to the second sequent and instantiates Y with x. For the first sequent **balance** applies again:

$$\begin{array}{lcl} hyp_1 & : & true(x,a)^- \\ hyp_2 & : & (xRy)^- \\ & \vdash & (yRx)^+ \end{array}$$

This sequent is now an instance of restriction b.

Example 14 $true(x, \Box a \supset \Box\Box a)$

This theorem belongs to modal logic $S4$ which requires R to be transitive (cf. restriction 4 in Section 9.5.1).

The plan starts with method **balance**. **D-reduction** compares the goal and the hypothesis and selects the following annotation:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Box a^-)^- \\ & \vdash & true(x, \boxed{\boxed{\Box(\Box a^+)^+_{\{c_1\}}}_{\{c_1\}}})^+ \end{array}$$

Since there is no wave rule applicable to the goal, this annotation would fail. The next possible annotation is:

$$\begin{array}{lcl} hyp_1 & : & true(x, \boxed{\boxed{\Box a^-_{\{c_1\}}}_{\{c_1\}}})^- \\ & \vdash & true(x, \boxed{\boxed{\Box(\Box a^+_{\{c_1\}})^+}_{\{c_1\}}})^+ \end{array}$$

Now rippling is possible in both the hypothesis and the goal. For the hypothesis rule **wr-**$\Box_e$ is used and for the goal rule **wr-**$\Box_i$ is applied twice. After rippling the goal once it looks like this:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Box a^-)^- \\ hyp_2 & : & \boxed{\boxed{\Pi_{y:W}(xRy)^+ \rightarrow true(y,a)^-_{\{c_1\}}}_{\{c_1\}}} \\ & \vdash & \boxed{\boxed{\Pi_{y_1:W}(xRy)^- \rightarrow true(y_1, \boxed{\boxed{(\Box a^+_{\{c_1\}})^+}_{\{c_1\}}})^+}_{\{c_1\}}} \end{array}$$

and after applying rule **wr-**$\Box_i$ again we obtain:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Box a^-)^- \\ hyp_2 & : & \boxed{\underline{\Pi_{y:W}(xRy)^+ \rightarrow true(y,a)^-}_{\{c_1\}}}_{\{c_1\}} \\ & \vdash & \boxed{\Pi_{y_1:W}(xRy)^- \rightarrow \Pi_{y_2:W}(y_1Ry_2)^- \rightarrow \underline{true(y_2,a)^+}_{\{c_1\}}}_{\{c_1\}} \end{array} \tag{9.16}$$

Method **weak-fertilise** applies and refines the goal with hypothesis hyp_2:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Box a^-)^- \\ hyp_2 & : & \Pi_{y:W}(xRy)^+ \rightarrow true(y,a)^- \\ hyp_3 & : & (xRy_1)^- \\ hyp_4 & : & (y_1Ry_2)^- \\ & \vdash & (xRy_2)^+ \end{array}$$

Again, this sequent is an instance of restriction 4 on the world-visibility relation.

Example 15 $true(x, \Diamond a \supset \Box \Diamond a)$

This is a theorem from modal logic S5. The world visibility restriction for this logic is 5.

To prove it, first balance:

$$\begin{array}{lcl} hyp_1 & : & true(x, \boxed{\underline{\Diamond a^-}_{\{c_1\}}}_{\{c_1\}})^- \\ & \vdash & true(x, \boxed{\underline{\Box(\Diamond \underline{a^+}_{\{c_1\}})^+}}_{\{c_1\}})^+ \end{array}$$

Then ripple the hypothesis with rule **wr-$\Diamond_e$**:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Diamond a^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y,a)^- \\ & \vdash & true(x, \underline{\boxed{\Box(\Diamond \underline{a^+}_{\{c_1\}})^+}}_{\{c_1\}})^+ \end{array}$$

then the goal with rule **wr-$\Box_i$**:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Diamond a^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y,a)^- \\ & \vdash & \boxed{\Pi_{z:W}(xRz)^+ \rightarrow true(z, \underline{\boxed{(\Diamond \underline{a^+}_{\{c_1\}})^+}}_{\{c_1\}})^+}_{\{c_1\}} \end{array}$$

and rule **wr-$\Diamond_i$**:

$$\begin{array}{lcl} hyp_1 & : & true(x, \Diamond a^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y,a)^- \\ & \vdash & \Pi_{z:W}(xRz)^- \rightarrow (zRY)^+ \end{array}$$

$$\begin{array}{lcl} hyp_1 & : & true(x, \diamond a^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y, a)^- \\ & \vdash & \boxed{\underline{\Pi_{z:W}(xRz)^- \rightarrow true(Y, a)^+}_{\{c_1\}}}_{\{c_1\}} \end{array}$$

Making meta-variable Y equal to y yields an axiom on the second sequent and leaves us with one single sequent to prove. **Balance** applies:

$$\begin{array}{lcl} hyp_1 & : & true(x, \diamond a^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y, a)^- \\ hyp_5 & : & (xRz)^- \\ & \vdash & (zRy)^+ \end{array}$$

Here, again, what's left is an instance of the restriction over the visibility relation.

Example 16 $true(x, \diamond\Box a \supset \Box \diamond a)$

The restriction on R for this theorem is 2 (*directedness*).

$$\forall xyz.xRy \wedge xRz \supset \exists w.yRw \wedge zRw$$

Balance:

$$\begin{array}{lcl} hyp_1 & : & true(x, \boxed{\underline{\diamond(\Box a^-_{\{c_1\}})^-}}_{\{c_1\}} \quad)^- \\ & \vdash & true(x, \boxed{\underline{\Box(\diamond a^+_{\{c_1\}})^-}}_{\{c_1\}} \quad)^+ \end{array}$$

Then ripple hypotheses:

$$\begin{array}{lcl} hyp_1 & : & true(x, \diamond(\Box a^-)^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y, (\Box a^-)^-)^- \\ hyp_4 & : & \boxed{\underline{\Pi_{z:W}(yRz)^+ \rightarrow true(z, a)^-_{\{c_1\}}}}_{\{c_1\}} \\ & \vdash & true(x, \boxed{\underline{\Box(\diamond a^+_{\{c_1\}})^+}}_{\{c_1\}} \quad)^+ \end{array}$$

and then ripple the goal:

$$\begin{array}{lcl} hyp_1 & : & true(x, \diamond(\Box a^-)^-)^- \\ hyp_2 & : & (xRy)^- \\ hyp_3 & : & true(y, (\Box a^-)^-)^- \\ hyp_4 & : & \Pi_{z:W}(yRz)^+ \rightarrow true(z, a)^- \\ & \vdash & \Pi_{z:W}(xRz)^- \rightarrow (zRY)^+ \end{array}$$

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \diamond(\Box a^-)^-)^- \\
hyp_2 & : & (xRy)^- \\
hyp_3 & : & true(y, (\Box a^-)^-)^- \\
hyp_4 & : & \boxed{\boxed{\Pi_{z:W}(yRz)^+ \rightarrow \underline{true(z,a)^-}_{\{c_1\}}}_{\{c_1\}}} \\
\vdash & & \boxed{\boxed{\Pi_{z:W}(xRz)^- \rightarrow \underline{true(Y,a)^+}_{\{c_1\}}}_{\{c_1\}}}
\end{array}
$$

Now, if we weak-fertilise the second goal with hyp_4, we will obtain:

$$
\begin{array}{lcl}
hyp_1 & : & true(x, \diamond(\Box a^-)^-)^- \\
hyp_2 & : & (xRy)^- \\
hyp_3 & : & true(y, (\Box a^-)^-)^- \\
hyp_4 & : & \Pi_{z:W}(yRz)^+ \rightarrow true(z,a)^- \\
hyp_5 & : & (x R \hat{z})^- \\
\vdash & & (\hat{z}RY)^+ \\
\vdash & & (yRY)^+
\end{array}
$$

This is again solved by appealing to the visibility condition.

9.6 Discussion

The N.D. rules for Modal logics we use here don't add any new problems as far as proof search is concerned. The new type of rule introduces two new hypotheses in the context simultaneously but this does not present any search problems.

There are other aspects to how our system has to be extended to deal with Modal logics. First we weak the notion of skeleton to allow the skeletons to be related by a relation different that equality, that is, relation R. This idea has been investigated elsewhere as relational rippling [Bundy & Lombart 95]. There, the idea is to adapt rippling to the context of logic programming.

The main problem with these logics is, as explained in Section 9.5.5, the rules for R. The neither inference rules nor the rewrite and wave rules obtained from them are sufficient for our system to finish the proofs. What is interesting though, is that the rest of the proofs are built leaving an instance of the appropriate restriction for R as the final sequent.

These sequnts need only to be recognised to be such instances for the proof to be finished. This can be done by a special purpose method without compromising the structure of the whole system.

9.7 Implementation

The system specification in Chapter 8 is a refinement of an earlier version. That version was implemented and some of the examples presented in this chapter were proven by that implementation.

The methods and submethods of the implementation specify tactics for a version of LF implemented in the Mollusc proof editor (Section 2.2). The proof planner on which the methods are used is MiniClam, a planner derived from Clam (Section 3.2) but logic-independent. Clam was designed for the Oyster logic (Section 2.2), therefore its mechanisms are closely linked to the syntax of this logic. MiniClam, on the other hand, has no link to any logic in particular, so different users can attach their own theories to the planner and, provided that the syntax in the methods and submethods is consistent, the planner will work for that theory.

MiniClam and Mollusc are linked by an interface that allows interaction between the two systems [Richards *et al* 94]. Once a proof plan has been constructed in MiniClam, it can be sent to Mollusc to be executed. The execution of the plan in Mollusc, if successful, will generate an actual proof of the theorem.

The tactic applied to the conjecture is built from the plan by the LF-interface connecting the planner and Mollusc. The tactic is meant to realise all the operations specified by the methods. Very often, the application of proof plans is successful and the proof is built.

In Appendix A, there is a description of the version of LF for Mollusc. It includes the specification of LF in Mollusc, the tactics available as well as the interface with MiniClam.

9.8 Summary and General Discussion

This section is split in two subsections. The first one contains a summary of the chapter. The second subsection has an overall analysis of the behaviour of the system in all the examples provided.

9.8.1 Summary

In this chapter we have analysed our system with some example logics and theorems. We first introduced propositional minimal logic and from there we augmented the rules to obtain more complex logics.

In minimal logic the method **unblock** was not used because all the inference rules translate into wave rules. We also saw an example in this logic where *cautious* balance is not appropriate and *eager* balance is required.

Minimal logic is at the core of other logics. We obtain these by adding new rules. In intuitionistic propositional logic, we add the rule for intuitionistic absurdity which can be translated into a rewrite rule but not as a wave rule. To apply this rule, method **unblock** is used. The application of this rule can be motivated by its enabling weak-fertilisation and so it is easy to control.

In classical propositional logic, the rule of *reductio ad absurdum* is used instead of the rule of intuitionistic absurdity. This rule is also applied as a rewrite by the method **unblock**. In this case, the rule cannot be motivated by weak-fertilisation as before and so its application is less constrained.

Predicate Logic requires Higher-Order Difference Unification in order to identify connections with predicate variables. We mentioned that introducing an explicit application operator allows us to use regular difference unification to hide arguments and keep the function symbols as part of the skeleton.

Wave rules for Predicate Logic introduce meta-variables in the rewritten expression. These rules are instantiated as the proof proceeds by unifying a goal and a hypothesis. This mechanism, however, does not trace typing dependencies and, in some cases, produces sequents whose contexts would not be typeable in the proof editor. We need to extend the system with a mechanism to prevent the instantiation of variables which are not typeable in LF. The problem of keeping track of dependencies in meta-variables is a well known one. Various unification algorithms for LF are aimed at solving this problem ([Pfenning 91],[Pym 90],[Dowek 91]). In the MRG group at Edinburgh, research into Higher-Order Difference Unification is currently being carried out.

Predicate Logic uses another non-standard type of rule: the *context* rewrite rule. This rule has a variable marked by an accent ˆ to indicate that, whenever the rule is applied, the marked variable will be instantiated by a new object level constant.

In Modal Logics, we assume the skeleton compatibility is modulo the visibility relation R. This kind of assumption is used in Relational Rippling ([Bundy & Lombart 95]). The main motivation for that research is to apply Rippling to proving properties of logic programs and circuits represented as relations.

In Section 9.7 we mentioned an implementation of an earlier version of the system. The differences of that implementation with the specification in Chapter refsystemdesign are listed in Section A.3.

Below we present a table that lists the theorems and logics used to test our methods:

Theorem	Logic
$(a \supset b) \wedge (b \supset c) \supset (a \supset c)$	min
$((a \vee c) \wedge (b \vee c)) \supset ((a \wedge b) \vee c)$	min
$a \supset \neg\neg a$	min
$(a \supset c) \wedge (b \supset c) \wedge (a \vee b) \supset c$	min
$(a \supset b \supset c) \supset (a \supset b) \supset (a \supset c)$	min
$((a \wedge b) \supset c) \supset a \supset b \supset c$	min
$((a \wedge b) \vee c) \supset ((a \vee c) \wedge (b \vee c))$	min
$((a \vee b) \wedge c) \supset ((a \wedge c) \vee (b \wedge c))$	min
$((a \wedge c) \vee (b \wedge c)) \supset ((a \vee b) \wedge c)$	min
$((a \wedge b) \wedge c) \supset (a \wedge (b \wedge c))$	min
$((a \vee b) \vee c) \supset (a \vee (b \vee c))$	min
$\neg\neg(\neg\neg a \supset a)$	int
$(\neg\neg a \supset \neg\neg b) \supset \neg\neg(a \supset b)$	int
$a \vee \neg a$	cla
$(a \supset b) \supset (\neg a \vee b)$	cla
$\forall p \supset \exists p$	pre
$\forall \lambda_{x:i}.(p(x) \supset q(x)) \supset \forall p \supset \forall q$	pre
$\forall p \supset \neg\exists\neg p$	pre
$\Box(a \supset b) \supset \Diamond a \supset \Diamond b$	K
$\Box(a \supset b) \supset (\Box a \supset \Box b)$	K
$\neg \Diamond \bot$	K
$\Diamond(a \vee b) \supset (\Diamond a \vee \Diamond b)$	K
$(\Diamond a \supset \Box b) \supset \Box(a \supset b)$	K
$\Box a \supset a$	T
$a \supset \Box \Diamond a$	B
$\Box a \supset \Box\Box a$	$S4$
$\Diamond a \supset \Box \Diamond a$	$S5$
$\Diamond\Box a \supset \Box \Diamond a$	$S2$

9.8.2 Discussion

Our system provides a framework for the analysis and development of proof techniques in various logics. Difference Reduction is the general strategy to follow for all the logics. The development of this strategy in Proof Plans provides also the possibility of implementing unblocking strategies specific to particular logics. Building Proof Plans also allows the inclusion of meta-variables to leave unsolved parts of the proof. These parts are instantiated as the proof plan is constructed but it is not always the case that the final instantiation will type check in the Proof Editor.

Our system design:

- Uses rewrite rules derived from logic presentations rather that inference steps from the framework logic.

- In the parts concerned with rewriting, works purely by difference reduction for logics where the inference rules can be translated into wave rules.

- Used the **unblock** method to help linking parts of a proof which are based on difference reduction. It works fine whenever only a few non-wave rule applications are needed to link such parts.

- Works less well for logics where meta-variables are introduced because we don't provide a mechanism to verify well-formedness at the object-level. The system design could easily be extended to include such mechanism by keeping track of the scope of the meta-variables or by using a typing routine.

- Does not work in general for logic presentations where some judgements are meant to make connections in the hypothesis list and not across the sequent symbol. This is the case of judgement R for the world visibility relation in the modal logics.

Our system shows that Proof Plans and Difference Reduction are useful to develop generic theorem provers applicable to logics with different characteristics. In our example logics, each one introduced different aspects to the process but these were not incompatible with what was already in use. Experience with more logics is needed to be able to make some judgement as to what kind of theories are suitable for our methods and which are not and also to find out if there is a complete set of extensions to our mechanisms that would account for well defined families of logics.

As we have seen in the extensions we have done to minimal logic, new logics need new extensions to the techniques. What is interesting, however, is that:

- Every extension is compatible with the previous machinery. Every development for a new logic deals with the properties of the new rules but is compatible with the tools already built for the rest of the rules.

- Every extension follows the pattern of Difference Reduction and is not necessarily

exclusive to one logic (e.g. meta-variables are used in Predicate Logic and in Propositional Modal logics).

We can see in the process described in this chapter that an incremental extension of logics required and incremental extension of proof techniques along axis of difference reduction. The method **unblock** groups the application of rules when a process is required outside the standard difference reduction procedure. We have identified two direct weakenings of the main strategy (i.e. application of plain rewrite rules and inference rules) but these need not be the only ways of solving parts of proofs which cannot be built by a difference reduction approach.

The **unblock** method can be seen as the "door" to other techniques rather than a default. A suitable **unblock** method could solve the problems posed by the world visibility relation rules in the modal logic presentations. Even if the method is not applicable to other logics, having an off-the-shelf framework to solve the main aspects of the proofs for a particular logic with a "port slot" (**unblock** method) where some ad-hoc details can be programmed is a good asset.

Chapter 10

Conclusions and Future Work

The development of framework logics has opened the possibility of representing formal systems in a uniform way. In these formalisms we can generalise current knowledge on proof automation to encompass a wide range of object theories. The idea is to develop proof automation techniques applicable to as many theories as possible.

Effective proof search in framework logics is a hard problem. It requires selecting appropriate rules from the framework logic as well as the instantiations for these corresponding to the object-level rules. The applicability of framework rules is high because they implement abstract operations independent of the object logic —like variable instantiation (e.g. $\rightarrow -elim$) or term generalisation (e.g. $\rightarrow -intro$)— and hence are applicable to a large number of object rules. Object level rules change from theory to theory and present different shapes and uses. This makes it difficult to abstract a method to account for the way they are all used.

In this thesis we have introduced a new approach to proof search in framework logics. The approach is based on difference reduction and proof plans. The contributions of our work stem from experience in designing and analysing a proof planning system for Natural Deduction style presentations of logics in the Edinburgh Logical Framework. We show that:

- Proof plans and difference reduction are promising paradigms to develop heuristics for proof search in framework logics.
- The extensions to the techniques of difference reduction we developed in this

thesis improve the power of the existing techniques and raise important issues for the development of the theory of difference reduction.

In the next two sections we review related work and address these two topics in detail.

10.1 Guiding Search by Difference Reduction in Framework Logics

Before analysing proof plans and difference reduction, we review some work done by other people and related to our own.

Research on proof search guidance in framework logics has focussed so far on uniform methods based on logic programming ideas [Helmink 91], [Pym 90], [Felty & Miller 91], [Pfenning 91], [Dowek 91]. These methods guide search by exploiting powerful unification algorithms suitable to type and higher order theories and extend resolution ideas to framework logics.

In [Helmink 91], the rules of both the meta and object levels are transformed into Horn clauses to enable a goal directed search. As the proof proceeds, new entries in the hypothesis list are dynamically transformed into Horn clauses too. This method provides a uniform treatment of framework and object-logic rules under a single Prolog style goal directed search method.

In [Felty 89], object level theories are encoded in a subset of Higher Order Logic (hh^{ω}) and tactical theorem proving based on λProlog [Miller & Nadathur 88] is proposed for some particular logics. In [Felty 91] LF signatures are translated into hh^{ω} to take advantage of the goal directed search mechanism already developed for that theory.

In the system Elf [Pfenning 91], LF constructors are given a direct operational interpretation. An extension of Higher Order Unification is given in order to cope with dependent function types.

[Pym 90] and [Dowek 91] also develop unification algorithm for type theories as basis for logic programming style search. Pym's work is on LF and Dowek's for the Calculus of Constructions.

The work just described take the first step towards endowing framework logics with an operational mechanism to guide search. [Helmink 91] transforms the inference rules encoded in a framework logic to suit a Prolog style search mechanism; [Felty 89] proposes the Prolog style representation and mechanism as the framework itself; [Pfenning 91] adds a goal directed mechanism to the type theory to preserve its declarative properties.

In [Helmink 91], the goal directed mechanism requires some tactical assistance to control search. In [Felty 89], as in Prolog-like systems, the *programming* of a signature is not independent of the search mechanism. It seems unavoidable to have to define clauses with operational predicates that disrupt the declarative nature of a signature and make the control mechanism ad-hoc to each logic.

Goal directed approaches to using signatures are difficult when these specify theories where introduction and elimination rules are involved. The problem is that elimination rules contain information related to the hypotheses and not to the goal. Any heuristic to apply them needs to analyse the hypothesis list to select the right elimination rule, even in a backwards proof. This makes uniform search paradigms such as resolution difficult to control.

Our approach is different. First, we constrain search at the framework level by using tactics that implement basic meta-level operations (e.g. rewriting, refinement, elimination and introduction). The methods that specify these tactics contain declarative heuristics in their preconditions that control their applicability. In this setting, the number of choices for the system (i.e. the planner) is far less than the free selection of meta-level rules. Another advantage of this approach, inherited from proof planning, is that heuristics are localised, explicit and declarative so they are easily understood and modified.

The extraction of rewrite rules from inference rules filters some undesirable structures like dummy variables (or place-holders) that make the reasoning about the proofs less uniform and more complicated.

At the object level, unification constrains the selection of rules like in the logic programming systems, but wave and polarity annotations provide information that constrain

even more the selection of rules; therefore, there is less search.

Our system uses the inference rules in three different levels of restriction. The most restricted level is wave rules. These are highly constrained and hardly generate any search. Not all rewrite rules can be parsed into wave rules, however. This means that some parts of the planning process would unavoidably fall into the second level of restriction: the level of the rewrite rules. These rules possess a similar degree of applicability to the rules used by the goal-directed search systems mentioned earlier but are more specific. Let us see what this means.

As we said before, controlling the application of elimination rules in goal-directed search systems is difficult. The head of the rules is applicable to nearly any goal because the connective is part of the body. In our rewrite rules, however, we have converted elimination rules into rewrites that apply to terms with negative polarity. The application of these rules is more constrained than the inference rules because the left-hand side of the rewrites contains the connective.

The third level of restriction of inference rules is the unrestricted one. The `unblock` method, in its last clause, applies inference rules directly. This feature enables the system to *degenerate gracefully* into brute force search and exploit the use of inference rules not contemplated by the rewrite and wave rule abstractions of them.

The `weak-fertilise` method in our system has a role equivalent to resolving a goal with a hypothesis or two hypotheses. The **d-reduction** method could be described as a kind of "resolution analysis" where difference unification and the measure of disagreement for annotated terms identify the most promising complementary terms to resolve and then use rippling to isolate them.

We concentrate on exploiting structural components of object logics to enhance the performance of the search guidance at the framework level. We combine forward and backward reasoning guided by annotations and polarity information.

Wave annotations and polarity are abstract concepts applicable generically but determined by the object theories. They constitute a way of linking general search guidance techniques with object-level components.

Our system explores the use of Rippling as difference reduction paradigm. The rest of the search is done by rewriting and inference rule application. These aspects of the search process left to the `unblock` method could probably become part of other difference reducing submethods, and thus reduce the uncontrolled search that the system has to do.

Our work indicates that difference reduction techniques like rippling reduce search considerably when they are applicable. We have experimented with rippling as a difference reduction technique and found that it reduces search considerably in parts of the search process when wave rules are involved. Our experience shows that:

- Difference reduction works in practice as a general strategy in proof search guidance and reduces search considerably when suitable difference reduction heuristics are used.
- Rippling can be used in non-inductive domains as a difference reduction heuristic. This extends significantly other work in this area.
- Logic presentations contain information that can be used to parameterise proof search guidance systems.

For Propositional logic we have used first order unification but for Predicate logic we needed second order unification. The implementation we have of these methods has been made in Prolog. To achieve the full power of our system, we will need to incorporate a more powerful unification algorithm. We conjecture this could be done taking advantage of λProlog's higher order unification as well as the experience of Felty [Felty 89] and Pfenning [Pfenning 91] with this language. It is possible too, to use some of the ideas in [Felty 92] to implement the rewriting system of our methods.

10.2 Extensions to Difference Reduction and Rippling

Difference Reduction was proposed as a general approach to automated reasoning drawing from experience on using Rippling to guide proof search on inductive theorems. In order to explore the extent to which these techniques can be used to guide search

generically, we have extended many concepts used until now in rippling and difference reduction to adapt them to our needs. We have used these extensions in the design and analysis of our methods but we have not explored all the consequences of these extensions in detail. Our extensions may constitute steps forward towards a unified theory of rippling where theoretical issues, like the ones addressed in Section 4.4, could be studied in more detail.

Nevertheless we consider that the extensions we have made are interesting on their own and should be pursued as topics of research. The extensions are the following:

- Sequent balancing.
- Coloured difference unification.
- Polarity annotations.
- Twin rules.

We now discuss these in detail.

10.2.1 Sequent Balancing

In inductive theorem proving there is no notion of balancing of sequents. The reduction of differences is done between induction hypotheses and conclusion and these are already *in place* when rippling is started.

When using rippling in domains other than inductive proofs we are likely to face the problem of identifying the terms to be d-unified. Balance is a heuristic to *prepare* a sequent for d-unification and rippling by drawing expressions across the sequent symbol. We have devised two strategies for this heuristic but there are many more.

When doing the cautious balance, only atomic subterms present in the goal are introduced. We don't verify that these atomic subterms occur with opposite polarity to ensure that they may lead to a connection. The reason for this is that in order to compute the polarity values of the subexpressions of a term, we do it with respect to annotations. Since balance is a process to prepare the sequent for d-unification, polarity annotations are not available at that stage.

If a more general notion of polarity is used, and polarity values are computed independently of wave annotations, the **balance** method could be made more accurate by tracing the movement of complementary expressions. See Section 10.2.3 below for the discussion on extending polarity.

10.2.2 Coloured Difference Unification

Coloured rippling involves coloured annotations (Section 3.3.1). Colours are assigned to distinguish overlapped annotations in the goal of a sequent that correspond to different induction hypotheses.

Earlier work in coloured rippling assigned colours by d-unifying the goal with several hypotheses and then trying to overlap all the resulting annotations of the goal. Each annotation marks a different skeleton and colours are assigned to distinguish them. See Section 3.3.1 for more details.

In this thesis, we have modified the difference unification algorithm to find all the possible common skeletons that can be annotated simultaneously in two expressions. The reason for this is that the skeletons contained in the goal might have their corresponding parts in one single hypothesis before they are eliminated later on in the proof. The simplest example of this is a conjunction in the hypothesis. It is important to identify as many skeletons as possible in the d-unification stage as we saw in Chapter 9.

Our algorithm naturally returns more solutions than difference unification. In our work we are interested in obtaining the maximum number of skeletons for the goal first but, for other systems, a different order of solutions may be required. The problem with Coloured Difference Unification algorithm is that it provides too many solutions and, in some cases, much of the search saved in terms of rule non-determinism is balanced by the trial-and-error of a large number of d-unifications. Studying different search strategies for the Coloured Difference Unification algorithm is an interesting and useful topic of research.

10.2.3 Polarity Annotations

In Clam (Section 3.2), polarity is computed to decide whether a wave rule is applicable to a term. The polarity values are computed with respect to a fixed logic (Oyster logic); they are transparent to the user and never marked on the expressions themselves.

In this thesis we have extended the use of polarity to identify connections. LF-level polarity values are computed to serve the same purpose they serve in Clam. Object-level polarity values are computed with respect to signatures to identify connections in the components of a sequent.

The concept of polarity is a rich one for automatic theorem proving.

An important property non-clausal resolution relies on is that in classical logic all atomic formulae can be assigned a polarity value. For non-classical logics this may not be the case in general but, we conjecture that it may still be possible to assign polarity values to some formulae with some combination of connectives. If this is possible, the reduction rules available in classical logic may not be available in other logics to reduce the expressions involving T and $\bot$. However, the complementary expressions may be identified and some mechanism may be available to *make* the connections (e.g. difference reduction, matrix systems, etc.).

Our polarity algorithm computes polarity values for some subexpressions of a term according to the wave rules extracted from a signature. The values computed depend on the particular presentation of the logic. We believe that polarity values are relative to the logic itself and so the presentation should not make a difference in the assignment of those values.

10.2.4 Rewriting as Rule Application

The idea used in inductive proofs of extracting rewrite rules and wave rules from equalities and implications can be extended to more complex structures, inference rules in particular. In order to incorporate as much as possible from the proof process into difference reduction, we have translated inference rules into rewrites by looking at the effects they produce in the proofs.

Many of the rules we obtained are not standard from the rewriting point of view. Our translation converts Horn-clause like rules into rewrite rules by using the concept of twin-rule. Twin-rules also allow us to simplify some inference rules like those with place-holder variables into a presentation more akin to rippling. Context-rules allow us to specify inference rule applications where new variables are introduced as rewrite rules. These rules, however, need a special mechanism to produce such variables when the rule is applied.

Still our method of translation of rules is not uniform. We have not contemplated all possible shapes of inference rules but only the most common ones. Future work on this aspect would involve finding a uniform representation of rewrite rules for all possible inference rule encodings in LF.

10.3 Extensions to the Design

Our system design can be improved in various ways. First of all we need to develop more heuristics to account for the tasks handled by the **unblock** method. These may be developed as new difference reduction techniques incorporated as new submethods or just as more elaborate preconditions for the application of rewrite and inference rules so that their application is more constrained.

In [Helmink 91], every new member of the context is dynamically translated into a Horn clause. In our system, rewrites are extracted from a signature before the planning is started and after that is fixed. We could automatically parse derived rules inserted in the context as rewrite and wave rules and added to the database in a way similar to Helmink's system.

10.4 Implementation

As we said before, the final design of the system in Chapter 8 in based on the experience obtained from an implementation of an earlier version of the methods and submethods in MiniClam.

We would certainly like to see the new version implemented in order to continue experi-

mentation with new theorems and logics. In order to cope with some of the deficiencies of the previous version and to account for new developments, we think a higher order logic programming language, —like λ-Prolog— would be better suited than Prolog for the task for reasons we have already expressed in this chapter.

Working with mathematical and logical formulae in ASCII terminal has always been difficult. Working with annotated formulae is harder. But working with coloured polarised annotated formulae at framework and object levels is, at times, almost impossible. Therefore, any future development of a system like the one described in this thesis should really involve some work on a good visual interface where terms can be easily read and edited.

A computer is such a wonderful medium to learn, experiment with and develop mathematics.

Bibliography

[Avron *et al* 87] A. Avron, F. Honsell, and I. Mason. Using typed lambda calculus to implement formal systems on a machine. Report ECS-LFCS-87-31, Department of Computer Science, University of Edinburgh, July 1987.

[Basin & Constable 93] D. Basin and R. Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, Cambridge, 1993. Cambridge University Press.

[Basin & Walsh 91] D. Basin and T. Walsh. Difference matching. Research Paper 556, Dept. of Artificial Intelligence, Edinburgh, 1991. To appear in the proceedings of CADE-11.

[Basin & Walsh 93] D. Basin and T. Walsh. Difference unification. In *Proceedings of the 13th IJCAI.* International Joint Conference on Artificial Intelligence, 1993. Also available as Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik.

[Basin & Walsh 94] D. Basin and T. Walsh. Termination orders for rippling. In Alan Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 466–83, Nancy, France, 1994. Springer-Verlag.

[Basin & Walsh 96] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 1996. To appear 16(2-3), 1996.

[Benthem L.S. 77] van Benthem L.S. *Checking Landau's "Grundlagen" in the Automath system.* Unpublished PhD thesis, Eindhoven University of Technology, 1977.

[Bibel 82] W. Bibel. *Automated Theorem Proving.* Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden, 1982.

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Bruijn 80] N.G. de Bruijn. A survey of the project Automath. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[Bundy & Lombart 95] A. Bundy and V. Lombart. Relational rippling: a general approach. In C. Mellish, editor, *Proceedings of IJCAI-95*, pages 175–181. IJCAI, 1995. Longer version to appear as a DAI research paper.

[Bundy *et al* 88] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. Research Paper 419, Dept. of Artificial Intelligence, Edinburgh, 1988. Also in the proceedings of IJCAI-89.

[Bundy *et al* 90a] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al* 90b] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Bundy *et al* 91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Caml light] INRIA. *The CAML LIGHT system : release 0.5 : documentation and user's manual.*

[Church 40] A. Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1):56–68, 1940.

[Constable & Howe 90] R.L. Constable and D.J. Howe. Nuprl as a general logic. In P. Odifreddi, editor, *Logic and Computer Science*, pages 77–90. Academic Press, 1990.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Coquand & Huet 85] Th. Coquand and G. Huet. Constructions: A higher order proof system for mechanising mathematics. In Springer, editor, *Proceedings of EUROCAL 85*, 1985.

[Coquand & Huet 88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Coquand & Paulin-Moring 88] T. Coquand and C. Paulin-Moring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *International Conference on Computer Logic*, volume LNCS 417 of *COLOG*, pages 50–66, December 1988.

[Coquand 86] Th. Coquand. An analysis of Girard's paradox. In *Proceedings of LICS.* IEEE, 1986.

[Curry & Feys 58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[Dowek 91] G. Dowek. *Démonstration Automatique dans le Calcul des Constructions.* Thèse de doctorat, Université Paris 7, décembre 1991.

[Felty & Miller 91] A. Felty and D. Miller. Encoding dependent type λ-calculus in an intuitionistic logic. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 215–251. Cambridge University Press, 1991.

[Felty 89] A. Felty. *Specifying and implementing theorem provers in a Higher Order Programming Language.* Unpublished PhD thesis, University of Pennsylvania, 1989.

[Felty 91] A Felty. Encoding dependant types in an intuitionistic logic. Technical Report 1521, INRIA, 1991.

[Felty 92] A. Felty. A logic programming approach to implementing higher-order term rewriting. In L-H Eriksson *et al*, editors, *Second International Workshop on Extensions to Logic Programming*, volume 596 of *Lecture Notes in Artificial Intelligence*, pages 135–61. Springer-Verlag, 1992.

[Gallier 86] J. Gallier. *Logic for Computer Science.* Harper & Row, New York, 1986.

[Gentzen 69] G. Gentzen. *The Collected Papers of Gerhard Gentzen.* North Holland, 1969. edited by Szabo, M.E.

[Gordon 88] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

[Gordon *et al* 79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation,* volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[Harper *et al* 92] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM,* 40(1):143–84, 1992. Preliminary version in LICS '87.

[Helmink 91] L. Helmink. Goal directed proof construction in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks.* CUP, 1991.

[Howard 80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism,* pages 479–490. Academic Press, 1980.

[Hughes & Cresswell 90] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic.* Routledge, 1990.

[Hutter & Kohlhase 95] D. Hutter and M. Kohlhase. A colored version of the λ-calculus. Seki-report sr-95-05, University of Saarland, 1995.

[Jouannaud *et al* 82] J.-P. Jouannaud, P. Lescanne, and F. Reinig. Recursive decomposition ordering and multiset orderings. Technical Report MIT/LCS/TM-219, MIT, June 1982.

[Klop 92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, vol 2,* volume 2, pages 1–116. Clarendon Press, Oxford, 1992.

[Luo & Pollack 92] Z. Luo and R. Pollack. Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992.

[Luo 89] Z. Luo. ECC: An Extended Calculus of Constructions. In Asilomar, editor, *Proc. of the Fourth Ann. Symp. on Logic in Computer Science,* California, U.S.A., 1989.

[Martin-Löf 79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for*

Logic, Methodology and Philosophy of Science, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[Martin-Löf 84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[Miller & Nadathur 88] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.

[Negrete 91] S. Negrete. Proof plans with hints. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, Edinburgh, 1991.

[Negrete 93] S. Negrete. Planes de prueba y demostración automática de teoremas con sigerencias. In C.F. Geyer and A.C. Costa, editors, *X Simpósio Brasileiro de Inteligência Artificial*, pages 93–111. Sociedade Brasileira de Computação, October 1993. Also available from Edinburgh as DAI Research Paper 661 (longer version).

[Nordström 93] B. Nordström. The ALF proof editor. In *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs*, 1993.

[Paulson 87] L.C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.

[Paulson 90] L.C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 77–90. Academic Press, 1990.

[Paulson 94] L.C. Paulson. *Isabelle: A generic theorem prover.* Springer-Verlag, 1994.

[Pfenning 91] F. Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 182. Cambridge University Press, 1991.

[Pym & Wallen 91] D. Pym and L.A. Wallen. Proof search in the λ-Π calculus. In *Logical Frameworks*, pages 309–340. CUP, 1991.

[Pym 90] D.J. Pym. *Proofs, search and computation in general logic.* Unpublished PhD thesis, University of Edinburgh, 1990. Available as LFCS report ECS-LFCS-90-125.

[Richards 93] B. L. Richards. Mollusc user's guide version 1.1. DAI Technical paper 23, University of Edinburgh, September 1993.

[Richards *et al* 94] B.L. Richards, I. Kraan, A. Smaill, and G.A. Wiggins. Mollusc: a general proof development shell for sequent-based logics. In A. Bundy, editor, *12th Conference on Automated Deduction*, pages 826–30. Springer-Verlag, 1994. Lecture Notes in Artificial Intelligence, vol 814; Also available from Edinburgh as DAI Research paper 723.

[Simpson 94] A.K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* Unpublished PhD thesis, Department of Computer Science, University of Edinburgh, 1994.

[Smaill & Green 96] A. Smaill and I. Green. Higher-order annotated terms for proof search. Research paper, Department of Artificial Intelligence, 1996. Submitted to the IX International Conference on Theorem Proving Higher Order Logics, Turku, Finland, August,1996.

[Smullyan 68] Raymond M. Smullyan. *First Order Logic.* Springer-Verlag, Berlin, 1968.

[vanHarmelen *et al* 93] Frank van Harmelen, Andrew Ireland, Andrew Stevens, Santiago Negrete, and Alan Smaill. The Clam proof planner, user manual and programmer manual (version 2.2). Technical report, DAI, 1993.

[Yoshida *et al* 94] Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. In A.G. Cohn, editor, *In proceedings of ECAI-94*, pages 85–89. John Wiley, 1994.

Appendix A

Mollusc LF

As described in Section 2.2, Mollusc is a generic proof editor where a particular logic can be specified by means of a set of files. A utility program provided by Mollusc uses this specification files to construct all the necessary structures to have a proof editor for the given logic [Richards 93].

The version of LF we used for the implementation is system G in [Pym & Wallen 91]. We chose this version because we need a version of the type system that gives access to both the goal and the hypotheses of a sequent, i.e a Sequent Calculus style presentation of LF. The first version of LF ([Harper *et al* 92]) is a natural deduction style presentation; all the rules in this version introduce new goals but no new hypotheses.

In Figure A.1 there is a list of the rules of system G. $\Gamma \vdash_\Sigma x : T$ states that it is provable in LF that object x inhabits type T from context Γ and signature Σ. $A =_{\beta\eta} A'$ means A and A' are equal modulo $\beta\eta$-reduction.

In Mollusc LF, a sequent consists of a context and a goal separated by the sequent symbol: ==>. The context is a list of typed constants and the goal is formed by a term of the form: $(Obj : Type)$ where $Type$ is a type representing the theorem we want to use the system to prove and Obj is the object that inhabits that type. This object will be a variable in the beginning of the proof and will be instantiated as the proof proceeds.

As mentioned in 2.1.1, LF uses signatures, that is, sets of typed constants that define object logics. In the Mollusc version of LF, a signature is embedded in the context of a sequent and its elements are distinguishable from the hypotheses by enclosing them in the function sig. Hence, every member of the signature will be a Prolog term of the form: $sig(Obj : Type)$.

In Mollusc LF, function applications have been made explicit by having an application operator: ^. The reason for this is that, since, only functional and dependent functional type constructors are allowed in LF, all functions are "curried" and Prolog terms do not allow partial function applications in its syntax. For example, in LF, implication is defined by the typed constant: $imp : o \rightarrow o \rightarrow o$. The term: $a \rightarrow b$ would have to look —in Prolog— like: $((imp(a))b$ which is not allowed. With the application operator, this term is represented in Prolog like: $((imp\hat{}a)\hat{}b)$.

$$\frac{\vdash_\Sigma \Gamma \ context \quad c : A \in \Sigma}{\Gamma \vdash_\Sigma c : A} \quad (ax1)$$

$$\frac{\vdash_\Sigma \Gamma \ context \quad c : A \in \Gamma}{\Gamma \vdash_\Sigma c : A} \quad (ax2)$$

$$\frac{\Gamma \vdash_\Sigma A : Type \quad \Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda_{x:A}.M : \Pi_{x:A}.B} \quad \Pi r$$

$$\frac{\Gamma \vdash_\Sigma A : Type \quad \Gamma, x : A \vdash_\Sigma M : B \quad x \notin FV(B)}{\Gamma \vdash_\Sigma \lambda_{x:A}.M : A \to B} \quad (\to r)$$

$$\frac{@ : \Pi_{x:A} B \in \Sigma \cup \Gamma \quad \Gamma \vdash_\Sigma N : A \quad B[N/x] =_{\beta\eta} C \quad \Gamma, y : C \vdash_\Sigma M : D}{\Gamma \vdash_\Sigma M[@N/y] : D} \quad (\Pi l)$$

$$\frac{@ : A \to B \in \Sigma \cup \Gamma \quad \Gamma \vdash_\Sigma N : A \quad \Gamma, y : B \vdash_\Sigma M : D}{\Gamma \vdash_\Sigma M[@N/y] : D} \quad (\to l)$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma A' : Type \quad A =_{\beta\eta} A'}{\Gamma \vdash_\Sigma M : A'} \quad (conv)$$

$$\frac{\Gamma, x : A \vdash_\Sigma X \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma X[N/x]} \quad (cut)$$

Figure A.1: Rules of LF's system G.

A.1 Tactics

LF's inference rules manipulate the constructs of the object theory merely as LF terms (i.e. typed constants and variables) but they don't individually correspond to any proof notion of the object theory. For example, using the LF signature for propositional logic in Figure A.2 to prove the following theorem:

$$\vdash \Pi_{a,b,c:o} true((a \supset b) \wedge (b \supset c)) \to true(a \supset c)$$

we can use LF Rule Πr backwards to introduce at the level of LF variables a,b,c and a hypothesis[1]:

$$\begin{array}{rcl} a & : & o \\ b & : & o \\ c & : & o \\ hyp & : & true((a \supset b) \wedge (b \supset c)) \\ & \vdash & \\ & & true(a \supset c) \end{array}$$

[1] The proof term is not displayed in the sequent.

$$
\begin{array}{rcl}
o & : & Type \\
true & : & o \rightarrow Type \\
\ldots, \wedge, \supset & : & o \rightarrow o \rightarrow o \\
 & \vdots & \\
\supset_i & : & \Pi_{A,B:o}(true(A) \rightarrow true(B)) \rightarrow true(A \supset B) \\
\supset_e & : & \Pi_{A,B:o}(true(A \supset B) \rightarrow true(A)) \rightarrow true(B) \\
\wedge_i & : & \Pi_{A,B:o} true(A) \rightarrow true(B) \rightarrow true(A \wedge B) \\
\wedge_{el} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(A) \\
\wedge_{er} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(B) \\
 & \vdots &
\end{array}
$$

Figure A.2: Section of a signature for Propositional Logic.

However, we might also want to apply Rule $\supset_i$ at the level of the objet logic (i.e. propositional logic). No LF rule corresponds to this object level operation, we need to apply several rules. We first need to find out that the term: $\supset_i\ (a)(c)$ has a type that matches the goal. Then we can use Rule Πl to eliminate $\supset_i$ with constants a and c to obtain a hypothesis that matches the goal[2].

Since, the user of the system would like to think about the problem at hand in terms of these proof notions of the object theory, we need to program them as tactics. Once these have been incorporated into the system, proofs will very seldom make use of direct calls to LF inference rules.

Below we describe the tactics we have developed for Mollusc LF:

A.1.1 Intro and Intros tactics

Intro applies Rule $\rightarrow r$,Πr, $ax1$, $ax2$ or $conv$. **Intros** tactic simply repeats **intro** as many times as possible.

A.1.2 Refine tactic

This tactic does a backward chaining operation of inference rules from the signature to the current goal.

The way it works is by applying (refining) a given constant (an inference rule from the signature) to a meta variable (Prolog variable), computing the type of the resulting object (which will include meta variables too) and match it with the goal. If the new term's type does not match, it is applied to a new meta variable to see if this time

[2] See the description of tactic **refine** below for a more detailed description of this process.

its type will match. This process continues until the type matches or the term is not typeable anymore.

This matching operation will give values to some of the meta variables introduced earlier. At this point, the tactic uses the Rule Πl to apply iteratively the original constant to the actual values obtained before for the metavariables. Each application will produce a new hypothesis which, in turn, will be applied to the next value, until all values have been exhausted. At the end of this process, a hypothesis matching the goal will have been produced ending that branch of the proof.

If matching the goal produces values for all the meta variables, no value of N in Rule Πl will go missing in the iteration and hence no subgoals will be left over. If, on the other hand, the matching leaves some meta variables without a value, the iteration of Rule Πl will leave their types as new subgoals. The overall effect of this tactic is that of backward chaining the given object inference rule to the goal as may be seen in the following example.

Let us pick up the example presented in the previous section. There we wanted to use Rule $\supset_i$ in Figure A.2 to refine the sequent. We can use *refine* tactic by calling: $refine{\supset_i}$. The tactic will first compute the type of $\supset_i$ and verify that it does not match the goal. Then, the next step will be to refine this constant by applying it to a meta variable: $\supset_i\ (MV_1)$. The type of this new term is:

$$\Pi_{B:o}(true(MV_1) \rightarrow true(B)) \rightarrow true(MV_1 \supset B)$$

Since this type does not match the goal, the tactic will refine again. The new term will be: $\supset_i\ (MV_1)(MV_2)$ whose type is: $(true(MV_1) \rightarrow true(MV_2)) \rightarrow true(MV_1 \supset MV_2)$. This type still does not match the goal so *refine* iterates once more over a new meta variable: MV_3. The rule applied this time will be Πl.

The new type $true(MV_1 \supset MV_2)$ matches the goal giving us back the variable assignment: $\{\ MV_1 = a, MV_2 = c\}$[3]. Meta variable MV_3 was not assigned any value because the final type *refine* obtained does not depend on it. This means that when rule $\rightarrow l$ was applied, N was not instantiated and it will be left as a goal.

After all this computation, the real application of rules begins. The tactic applies Rule Πl with $@ = \supset_i$ and $N = MV_1$. Now, the sequent will look like this[4]:

$$\begin{array}{rcl} a & : & o \\ b & : & o \\ c & : & o \\ hyp & : & true((a \supset b) \wedge (b \supset c)) \\ hyp_1 & : & \Pi_{B:o}(true(MV_1) \rightarrow true(B)) \rightarrow true(MV_1 \supset B) \\ & \vdash & \\ & & Obj : true(a \supset c) \end{array}$$

[3] The meta variables MV_1 and MV_2 are assigned type o and throughout the process this type is verified against any value for the variable.

[4] In the following sequents, the variables for the objects inhabiting the goals are displayed

Refine repeats the same procedure for the rest of the meta variables; in this case:MV_2 and MV_3. First, Rule Πl will be applied with $@ = hyp1$ and $N = MV_2$:

$$\begin{array}{rcl} a & : & o \\ b & : & o \\ c & : & o \\ hyp & : & true((a \supset b) \wedge (b \supset c)) \\ hyp_1 & : & \Pi_{B:o}(true(MV_1) \rightarrow true(B)) \rightarrow true(MV_1 \supset B) \\ hyp_2 & : & (true(MV_1) \rightarrow true(MV_2)) \rightarrow true(MV_1 \supset MV_2) \\ & \vdash & \\ & & Obj : true(a \supset c) \end{array}$$

Then, Rule $\rightarrow l$ will be applied with $@ = hyp_2$ and $N = MV_3$:

$$\begin{array}{rcl} a & : & o \\ b & : & o \\ c & : & o \\ hyp & : & true((a \supset b) \wedge (b \supset c)) \\ hyp_1 & : & \Pi_{B:o}(true(MV_1) \rightarrow true(B)) \rightarrow true(MV_1 \supset B) \\ hyp_2 & : & (true(MV_1) \rightarrow true(MV_2)) \rightarrow true(MV_1 \supset MV_2) \\ hyp_3 & : & true(MV_1 \supset MV_2) \\ & \vdash & \\ & & Obj : true(a \supset c) \\ & & MV_3(true(MV_1) \rightarrow true(MV_2)) \end{array}$$

At this point, tactic intro is called and meta variables MV_1 and MV_2 will be instantiated; Obj will be assigned hyp_3 and MV_3 will be left as a new goal:

$$\begin{array}{rcl} a & : & o \\ b & : & o \\ c & : & o \\ hyp & : & true((a \supset b) \wedge (b \supset c)) \\ hyp_1 & : & \Pi_{B:o}(true(a) \rightarrow true(B)) \rightarrow true(a \supset B) \\ hyp_2 & : & (true(a) \rightarrow true(c)) \rightarrow true(a \supset c) \\ hyp_3 & : & true(a \supset c) \\ & \vdash & \\ & & MV_3 : (true(a) \rightarrow true(c)) \end{array}$$

The overall effect of this tactic application was to chain backwards Rule $\supset_i$ in signature A.2 with goal $true(a \supset c)$.

A.1.3 Elim tactic

Elim tactic chains forwards two hypotheses producing a new one. This tactic takes

two hypothesis and tries to apply the first one to the second one. If this is not possible, tries to refine (in the same way of **refine** tactic above) the first hypothesis by applying it to meta variables until the result can be chained forwards with the second hypothesis using Rule $\rightarrow l$. For example, we could call $elim(\wedge_e l, hyp)$ on the last sequent in the last example, to eliminate $\wedge$ in hypothesis *hyp*. **Elim** will introduce meta variables as before until it finds that $\wedge_{el}(MV_1)(MV_2)$ has the right type: $true(MV_1 \wedge MV_2)$ (where $MV_1, MV_2 : o$). After this, Rule $\rightarrow l$ is applied to obtain the desired sequent:

$$
\begin{array}{rcl}
a & : & o \\
b & : & o \\
c & : & o \\
hyp & : & true((a \supset b) \wedge (b \supset c)) \\
hyp_1 & : & \Pi_{B:o}(true(a) \rightarrow true(B)) \rightarrow true(a \supset B) \\
hyp_2 & : & (true(a) \rightarrow true(c)) \rightarrow true(a \supset c) \\
hyp_3 & : & true(a \supset c) \\
hyp_4 & : & \Pi_{B:o} true((a \supset b) \wedge B) \rightarrow true(a \supset b) \\
hyp_5 & : & true((a \supset b) \wedge (b \supset c)) \rightarrow true(a \supset b) \\
hyp_6 & : & true(a \supset b) \\
 & \vdash & \\
 & & (true(a) \rightarrow true(c))
\end{array}
$$

A.1.4 Rewrite tactic

This tactic is more powerful than **refine** or **elim**. It rewrites a subjudgement of a hypothesis or the goal using a rule in the context as a rewrite rule in the given direction.

The tactic takes as arguments a judgement, a hypothesis name or the word "goal", an inference rule name and a direction. The judgement must be a subjudgement of the hypothesis indicated or the goal, as the case may be.

If a hypothesis is given, the sequent or sequents obtained after the tactic is applied, will have new hypotheses reflecting the rewriting of the hypothesis indicated with the rule. The rewriting works as we have described it for the planning level in Chapter 7. Equally, as in the planning level, when the goal is selected for rewriting, it is replaced by a new goal reflecting the rewriting.

The tactic works by cutting in a formula that represents the desired rewriting. This will introduce two new sequents: one that contains that new formula cut in as a new hypothesis and one where the formula cut in has to be proven (justified).

If the goal is being rewritten, refining the goal of the first new sequent delivers the desired outcome of the tactic. If a hypothesis is being rewritten, applying intro tactic to the first new sequent produces the desired outcome. In any case, the second new sequent, the justification, is proven automatically by the tactic.

For example, in the sequent:

$$\vdash true(a \wedge b) \rightarrow true(c)$$

the judgement $true(a \wedge b)$ can be rewritten with rule $land_{el}$ from the signature in Figure A.2 as a rewrite rule applied left-to-right. The first step is to cut in a formula that reflects the desired effect. In this case, the formula is: $(true(a) \rightarrow true(c)) \rightarrow (true(a \wedge b) \rightarrow true(c))$. When cut in with *cut* rule, two new sequents are produced:

$$\begin{array}{rcl} hyp_1 & : & (true(a) \rightarrow true(c)) \rightarrow (true(a \wedge b) \rightarrow true(c)) \\ & \vdash & true(a \wedge b) \rightarrow true(c) \end{array}$$

$$\vdash (true(a) \rightarrow true(c)) \rightarrow (true(a \wedge b) \rightarrow true(c)) \qquad \text{(A.1)}$$

Refining the goal of the first sequent with hyp_1 produces the desired outcome of the tactic:

$$\begin{array}{rcl} hyp_1 & : & (true(a) \rightarrow true(c)) \rightarrow (true(a \wedge b) \rightarrow true(c)) \\ & \vdash & (true(a) \rightarrow true(c)) \end{array}$$

Sequent A.1 is proven automatically by the tactic using rule $\wedge_{el}$.

As an example of rewriting a hypothesis, let's take the following sequent:

$$\begin{array}{rcl} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ & \vdash & true(d) \end{array}$$

To rewrite $true(a \wedge b)$ in hyp_1 with rule $\wedge_i$ applied right-to-left, the rewrite tactic cuts in the formula: $((true(a) \rightarrow true(c)) \rightarrow (true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d)$. The two new sequents are:

$$\begin{array}{rcl} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ hyp_2 & : & ((true(a) \rightarrow true(c)) \rightarrow true(d)) \rightarrow ((true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d) \\ & \vdash & true(d) \end{array}$$

$$\begin{array}{rcl} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ & \vdash & ((true(a) \rightarrow true(c)) \rightarrow true(d)) \rightarrow ((true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d) \end{array} \qquad \text{(A.2)}$$

Refining the goal of the first sequent with hyp_1 two new sequents are obtained:

$$\begin{array}{lll} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ hyp_2 & : & ((true(a) \rightarrow true(c)) \rightarrow true(d)) \rightarrow ((true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d) \\ & \vdash & (true(a) \rightarrow true(c)) \rightarrow true(d) \end{array}$$

$$\begin{array}{lll} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ hyp_2 & : & ((true(a) \rightarrow true(c)) \rightarrow true(d)) \rightarrow ((true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d) \\ & \vdash & (true(b) \rightarrow true(c)) \rightarrow true(d) \end{array}$$

Finally, introducing $\rightarrow$ in both sequents returns the final outcome sequents for the tactic:

$$\begin{array}{lll} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ hyp_2 & : & ((true(a) \rightarrow true(c)) \rightarrow true(d)) \rightarrow ((true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d) \\ hyp_3 & : & true(a) \rightarrow true(c) \\ & \vdash & true(d) \end{array}$$

$$\begin{array}{lll} hyp_1 & : & true(a \wedge b) \rightarrow true(c) \\ hyp_2 & : & ((true(a) \rightarrow true(c)) \rightarrow true(d)) \rightarrow ((true(b) \rightarrow true(c)) \rightarrow true(d)) \rightarrow true(d) \\ hyp_3 & : & true(b) \rightarrow true(c) \\ & \vdash & true(d) \end{array}$$

Again, the justification sequent (Sequent A.2) is proven automatically by the tactic.

A.2 MiniClam

MiniClam is a version of Clam independent from Oyster. MiniClam will work for any theory provided that the syntax of the methods available is consistent with the syntax of the theory.

Mollusc provides an interface to invoke a planning program on its sequents. We have used this interface to like the LF version in Mollusc and MiniClam. The syntax of the implemented methods for the first version of the system is similar to system G (Figure A.1) but the rules assumed do not involve building an object of the type representing the theorem. The system with this characteristic is called *system L* [Pym & Wallen 91] and is explained below. System L and system G are equivalent as far a provability is concerned [Pym 90].

In Figure A.3 there is a list of the rules of system L. The symbol $\vdash_\Sigma$ denotes deducibility in LF using signature Σ. $G\backslash cut$ refers to system G (Figure A.1) without the cut rule.

A.2.1 Mollusc-MiniClam interface

Mollusc and MiniClam are linked together via the planner interface provided by Mollusc. The interface consists of a Prolog file where all the commands of the proof planner

$$(Ax1)\quad \Gamma, x : A, \Gamma' \vdash_\Sigma A$$

$$(Ax2)\quad \Gamma \vdash_{\Sigma, c:A, \Sigma'} A$$

$$(\rightarrow r)\quad \frac{\Gamma \vdash_\Sigma B}{\Gamma \vdash_\Sigma A \rightarrow B} \qquad (a)\ x \notin Dom(\Gamma)$$

$$(\Pi r)\quad \frac{\Gamma, x : A \vdash_\Sigma B}{\Gamma \vdash_\Sigma \Pi_{x:A} B} \qquad (a)\ x \notin Dom(\Gamma)$$

$$(\rightarrow l)\quad \frac{\Gamma \vdash_\Sigma A \quad \Gamma, y : B \vdash_\Sigma C}{\Gamma \vdash_\Sigma C} \qquad \begin{array}{l}(a)\ @ : A \rightarrow B \in \Sigma \cup \Gamma \\ (b)\ y \notin Dom(\Gamma)\end{array}$$

$$(\Pi l)\quad \frac{\Gamma, y : D \vdash_\Sigma C}{\Gamma \vdash_\Sigma C} \qquad \begin{array}{l}(a)\ @ : \Pi_{x:A} B \in \Sigma \cup \Gamma \\ (b)\ y \notin Dom(\Gamma) \\ (c)\ G \backslash cut\ proves\ \Gamma \vdash_\Sigma N : A \\ (d)\ B[N/x] \rightarrow_{\beta\eta} D\end{array}$$

Figure A.3: Rules of LF's system L.

accessible to Mollusc are declared. This file also specifies some transformations on the sequent to be transferred from Mollusc to MiniClam. This transformations are called *preprocessing.*

First, the object-proof disappears because it is not used at the planning level. The planner uses LF system L (Figure refsystem-l). Second, the explicit application operator is removed from the goal and hypothesis (not from the signature) by "uncurrying" the expressions, using the typing information, to form simple Prolog terms. The third change that undergoes the original sequent is the addition of polarity values at both LF and object level. When a sequent is sent to the planner, a data base of rewrite and wave rules is also created from the signature.

A.3 First Version of the Methods

As we mentioned in Section 9.7, we have an implementation of an early version of the methods described in Chapter 8. This implementation was used to run examples and obtain enough information to produce the new version. The implementation differs from the new version in that:

1. It has only eager balance (first version of Balance).
2. It does not use coloured difference unification but *monochromatic* difference unification (for polarised terms). As a consequence, all weakenings of terms are rules done by the methods of the current version are not implemented in the old one.

3. Annotations in hypotheses are not removed as they are rippled, like in the new version. The hypotheses that have to be rippled are tagged so that the methods can identify them.

These differences explain the difference in performance outlined in the table shown in Section 9.8.1.